目錄

JavaScript 轻量级函数式编程

> 来源：ikcamp/Functional-Light-JS

沪江Web前端团队

> 参与者（排名不分先后）：阿希、blueken、brucecham、cfanlife、dail、kyoko-df、l3ve、lilins、LittlePineapple、MatildaJin、冬青、pobusama、Cherry、萝卜、vavd317、vivaxy、萌萌、zhouyao
>
> 关于译者：这是一个流淌着沪江血液的纯粹工程：认真，是 HTML 最坚实的梁柱；分享，是 CSS 里最闪耀的一瞥；总结，是 JavaScript 中最严谨的逻辑。经过捶打磨练，成就了本书的中文版。本书包含了函数式编程之精髓，希望可以帮助大家在学习函数式编程的道路上走的更顺畅。比心。

本书主要探索函数式编程[1](FP)的核心思想。在此过程中，作者不会执着于使用大量复杂的概念来进行诠释，这也是本书的特别之处。我们在 JavaScript 中应用的仅仅是一套基本的函数式编程概念的子集。我称之为“轻量级函数式编程(FLP)”。

注释： 题目中使用了“轻量”二字，然而这并不是一本“轻松的”“入门级”书籍。本书是严谨的，充斥着各种复杂的细节，适合拥有扎实 JS 知识基础的阅读者进行研读。“轻量”意味着范围缩小。通常来说，关于函数式编程的 JavaScript 书籍都热衷于拓展阅读者的知识面，并企图覆盖更多的知识点。而本书则对于每一个话题都进行了深入的探究，尽管这种探究是小范围进行的。

让我们面对这个事实：除非你已经是函数式编程高手中的一员（至少我不是！），否则类似“一个单子仅仅是自函子中的幺半群”这类说法对我们来说毫无意义。

这并不是说，各种复杂繁琐的概念是无意义的，更不是说，函数式编程者滥用了它们。一旦你完全掌握了轻量的函数式编程内容，你将会／但愿会想要对函数式编程的各种概念进行更正式更系统的学习，并且你一定会对它们的意义和原因有更深入的理解。

但是我更想要让你能够现在就把一些函数式编程的基础运用到 JavaScript 编程过程中去，因为我相信这会帮助你写出更优秀的，更符合逻辑的代码。

更多关于本书背后的动机和各种观点讨论，请参看前言。

本书

目录

关于出版

本书主要在 on Leanpub 平台上以电子版本的形式进行出版。我也尝试出售本书的纸质版本，但没有确定的方案。

除了购买本书以外，如果你想要对本书作一些物质上的捐赠，请在 patreon 上进行操作。本书作者感谢你的慷慨解囊。

Support my work on patreon

真人教学课程

本书内容大多源自于我教授的一个同名课程（以公司举办的公开或内部研讨会这样的形式进行）。

http://getify.me

如果你喜欢本书的内容，并希望组织此类课程，或者组织关于其他 JS／HTML5／Node.js 课程，请通过以下方式联系我： http://getify.me

在线视频课程

我还提供一些可以在线点播的 JS 培训课程。我在 Frontend Masters 上开办课程，例如我的 Functional-Lite JS 研讨会。还有一些课程发布在 PluralSight 上。

Contributions

关于内容贡献

非常欢迎对于本书的任何内容贡献。但是在提交 PR 之前请务必认真阅读 Contributions Guidelines。

License & Copyright

版权

1. FP，本书统称为函数式编程。

JavaScript 轻量级函数式编程

序

众所周知，我是一个函数式编程迷。我尝试阅读最新的学术论文，业余时间乃至工作间隙研究抽象代数（译者注：抽象代数又称近世代数，是研究各种抽象公理化代数系统的数学学科，也是现代计算机理论基础之一），并四处传播函数式编程的理念和语言。我所书写的 JavaScript 代码，每一条语句都是纯的。没错，我就是一个彻头彻尾的函数式编程教条式的狂热者。关于为什么要写纯的语句，请看我写的这本书。

其实我以前并不是这样子... 我曾痴迷于面向对象，并热衷于使用面向对象的方法来构建"真实世界"。我是人造机器人的发明者，夜以继日地修正机器人以达到更高精度的控制力。我也是有意识木偶的创造者，手指在键盘上的轻舞飞扬赋予了它们生命。做为黑客界的盖比特（译者注：盖比特是玩具之父），在连续不间断的写了 5 年面向对象的代码后，我对于这些成果还是不甚满意。整个过程也并不顺利，我一直感觉自己是一个糟糕的程序员，甚至失去了信心，认为写出既简单，又灵活同时又很好扩展的代码是不可能的。

我想是时候去尝试一些新的方法了，我开始涉足函数式编程的理念，并把它用在我的代码中。我的同事对此非常惊诧，他们根本不知道我在干什么。那段时间里我写的代码非常糟糕、另人生厌、简直是垃圾。造成这样结果的原因是我缺少一个目标或者说愿景。当然现在那个会编码的蟋蟀杰明尼（译者注：原文使用 Jiminy-Coding-Cricket 迪士尼动画人物蟋蟀杰明尼来暗指之前蹩脚的自己）已经不在了。在花费了好长时间，写了好多垃圾程序后我才弄明白怎样正确进行函数式编程。

现在，经历了那些乱七八糟的探索后，我感觉到纯函数编程实现了它所承诺的代码可读性和可复用。我不再发明而是发现我的模型，我像一个正在揭开巨大阴谋的侦探，在软木板上钉满了数学证据。一个数字时代的库斯托（译者注：库斯托是个传奇式的人物，探险家、电影制片人，一个享有戴高乐将军一样世界性声誉的法国人，作者比喻自己学习函数式编程就像库斯托探索海洋一般）以科学的名义记录下了这片奇特土地的特征！虽然并不完美，仍有很多东西要学习，但我对我的工作和产出从未有过现在这般满意！

假如一开始就有这本书，我探索纯函数式编程世界的道路就会更平坦一点，而不是荆棘满地。本书有两层：第一层教会你如何在每天的编码工作中，有效地使用各种各样的函数式构造方法。另一层则更重要，本书会提供一个准星，确保你不会偏离函数式编程的原则。

函数式编程是一种编程范式，Kyle 倡导使用它来实现声明式编程和函数式编程，同时该范式还可以与 JavaScript 世界形成平衡和互动。通过学习本书，你无需彻底理解范式的一切，就能了解纯函数式编程的基础；你无需重新创造轮子，就能获得练习和探索函数式编程的技

能，并让代码运行良好；你无需像我之前一样漫无目的地徘徊、甚至走回头路就能让你的职业生涯更上一层楼。你的合作者和同事们一定会欣喜若狂！

Kyle （译者注：Kyle 是火爆全球的《你不知道的 JavaScript》一书原作者）是一位伟大的老师，他对函数式编程的宏伟蓝图不懈追求，不放过任何一个角落和缝隙，同时他也苦学习者之苦。他的风格与行业产生共鸣，将大家的水平整体提高了一个档次。他的工作成果不仅出现在很多人的收藏夹中，也在 JavaScript 发展历史上占据坚实地位。Kyle 老师是绝世高手，你值得拥有。

函数式编程有很多种定义。Lisp 程序员和 Haskell 程序员对于函数式编程的定义截然不同。OCaml 和 Erlang 语言对于函数式编程范式的看法也大相径庭。即使在同一种语言 JavaScript 中，你也能看到函数式编程不同的定义。但总有一种纽带把这些不同的函数式编程连接在一起，这个纽带是一个有些模糊的"我一看就知道"的定义，这听起来有点下流（有人确实觉得函数式编程下流）。本书旨在抓住这个纽带，并不让你学习某些圈子的固定习语，而是让你获取相关知识，这些知识不论在哪个语言的函数式编程中都适用。

本书是你开启函数式编程旅途的绝佳起点。开始吧，Kyle 老师...

-Brian Lonsdorf (@drboolean)

JavaScript 轻量级函数式编程

前言

> 单子是自函子范畴上的一个幺半群

有晕头转向吗？不要担心，我自己也被绕晕了！对于那些已经了解函数式编程的人来说，这些专业术语才有意义，然而对于大部分人而言，它们没有任何意义。

这本书并不打算教你以上那些专业术语的具体含义。如果那正是你想查找的，请继续查阅。事实上，已经有很多从头到尾（正确的方式）介绍函数式编程的书了。如果你在深入学习函数式编程，这些专业术语有很重要的意义，你肯定会对这些专业术语越来越熟悉。

但是本书打算以另一种方式讲解函数式编程。我将从函数式编程的一些基础概念讲起，并尽可能少用晦涩难懂的专业术语。我们将尝试以更实用的方法来探讨函数式编程，而非纯粹的学术角度。毫无疑问，肯定会有专业术语。但是我将会小心谨慎的引入这些术语并解释为何它们如此重要。

可悲的是我并非酷酷的函数式编程俱乐部的一员。我从没有正式学过函数式编程。尽管我有计算机方面的教育背景并对数学有一定了解，但数学符号跟我理解的编程完全是两回事。我从来没写过一行 Scheme、Clojure 或 Haskell 代码，也不是老派的 Lisp 程序员。

我曾参加过不计其数的讨论函数式编程的会议，每次都希望能彻底搞明白函数式编程中那些神秘的概念到底是什么意思。然而每次我都失望而归，那些概念在我脑海里乱成一团，我甚至不清楚自己学了些什么。也许我学到了些东西吧，但是很长时间以来我都不能确定自己学到了什么。

通过不断的编程实践，而非站在学术的角度，我慢慢的理解了那些对函数式编程者[1]来说很简单直白的重要概念。你是否也有类似的经历 —— 你早就知道一件事，但直到很久之后你突然发现它竟然还有一个你从来不知道的名字！？

也许你像我一样；好几年前就听说过像“map-reduce”，“big data”等这些术语，但并不懂它们的实际意义。最终我明白了map(..)函数到底做了哪些事情 —— 在我知道列表操作是通向函数式编程者之路的基石，并且为何它们如此重要之后。我知道映射很久了，甚至在我知道它叫 `map(..)` 之前。

最终我开始整理这些想法并将它们称之为「轻量级函数式编程」（FLP）。

使命

但是，为什么学习函数式编程如此重要，即便只是学习轻量级函数式编程？

最近几年我越来越深刻的理解到编程的核心是人，而不是代码，我甚至将其视为一种信仰。我坚信代码只是人类交流的手段，只是它产生的副作用（仿佛听到了自我引用的笑声）才对电脑发出具体指令。

在我看来，函数式编程的核心在于让你在编程时使用一些广为人知、易于理解的模式。经过验证，这些模式可以有效隔离让代码难以理解的错误。所以，函数式编程 —— 咳，轻量级函数式编程 —— 是每个开发者都可以掌握的重要工具之一。

> monad的含义是，一旦你搞懂了，你就无法跟别人解释什么是monad了。
>
> Douglas Crockford 2012 "Monads and Gonads"
>
> https://www.youtube.com/watch?v=dkZFtimgAcM

我希望这本书有可能打破上面的诅咒，尽管我们要到最后的附录部分才开始讨论「monad」。

科班出身的函数式编程者经常宣称只有 100% 使用函数式编程才算是真正地使用函数式编程：这是一种要么全有要么全无的主张。它会让人觉得如果编程时只有一部分使用了函数式编程而另一部分没用到，整个程序会被那些没有使用函数式编程的部分污染，从而认为使用函数式编程并不值得。

我想明确地说：我认为绝对主义并不存在。这没有意义，就像愚蠢地建议我只有使用完美的语法，这本书才算完美，如果犯了一点点错误，就会让整本书质量变低一样。

我写地越清楚，前后越一致，你阅读此书的体验将越好。但我不是一个完美无缺的作者。有些章节可能比另外一些写的好。但是那些有待提高的章节不会使书中写的好的部分黯然失色。

同样的道理也适用于代码。随着你越来越多的使用函数式编程的模式，你的代码质量会越来越高。25% 的时间使用它们，你会得到一些好处。80% 的时间使用它们，你将收益更多。

除了几处仅存的特例，你不会在本书里看到很多绝对的论断。我们讨论的是要追求的目标和现实中方方面面的权衡。

欢迎来到最实用的函数式编程的学习之旅。我们将共同探讨学习！

1. FPer，本书统称为函数式编程者。

JavaScript 轻量级函数式编程

第 1 章：为什么使用函数式编程？

> 函数式编程人员：没有任何一个函数式编程者会把变量命名为 x，函数命名为 f，模块代码命名为“zygohistomorphic prepromorphism”。
>
> James Iry @jamesiry 5/13/15
>
> https://twitter.com/jamesiry/status/598547781515485184

函数式编程（FP），不是一个新的概念，它几乎贯穿了整个编程史。我不确定这么说是否合理，但是很确定的一点是：直到最近几年，函数式编程才成为整个开发界的主流观念。所以我觉得函数式编程领域更像学者的领域。

然而一切都在变。不只是从编程语言的角度，一些库和框架都对函数式编程的兴趣空前高涨。你很可能也在读相关内容，因为你终于意识到函数式编程是不容忽视的东西。或者你跟我一样，已经尝试很多次去学函数式编程，但却很难理解所有的术语或数学符号。

无论你出于何目的翻阅本书，欢迎加入我们！

置信度

我有一个非常简单的前提，这是我作为软件开发老师（JavaScript）所做的一切基础：你不能信任的代码是你不明白的代码。此外，对你不信任或不明白的代码，你将不能确定这些代码是否符合你的业务场景。代码运行时也只能祈求好运。

信任是什么意思？信任是指你通过读代码，不仅是跑代码，就能理解这段代码能干什么事，而不只是停留在它可能是干什么的层面。也许我们不应该总倾向于通过运行测试程序，来验证程序的正确性。我并不是说测试不好，而是说我们应该对代码了如指掌，这样我们在运行测试代码之前就会知道它肯定能跑通。

通过读代码就能对我们的程序更有信心，我相信函数式编程技术的基础构成，是本着这种心态设计的。理解函数式编程并在程序中用心实践的人，得益于函数式编程已经被证实的原则，能够写出可读性高和可验证的代码，来达到他们想要的目的。

我希望你能通过理解轻量级函数式编程的原则，对你编写的代码更有信心，并且能在之后的路上越走越好。

交流渠道

函数式编程为何如此重要？为了回答这个问题，我们退一万步先来讨论一下编程本身的重要性。

我认为代码不是电脑中的一堆指令，这么说你可能感到很奇怪。事实上，代码能指示电脑运行就是一个意外的惊喜。

我深信代码的主要作用是方便人与人交流。

根据以往经验你可能知道，有时候花很多时间“编程”其实只是读现有的代码。我们的大部分时间其实都是在维护别人的代码（或自己的老代码），只有少部分时间是在敲新代码。

你知道研究过这个话题的专家给出了怎样的数据吗？我们在维护代码过程中 70% 的时间花在了阅读和理解代码上。 也难怪全球程序员每天的平均代码行数是 5 行。我们一天花七个半小时用来读代码，然后找出这 5 行代码应该写在哪里。

我想我们应该更多的关注一下代码的可读性。可能的话，不妨多花点时间在可读性上。顺便提一句，可读性并不意味着最少的代码量，对代码的熟悉程度也会影响代码的可读性（这一点也是被证实过的）。

因此，如果我们要花费更多的时间来关注代码的可读性和可理解性，那么函数式编程为我们提供了一种非常方便的模式。函数式编程的原则是完善的，经过了深入的研究和审查，并且可以被验证。

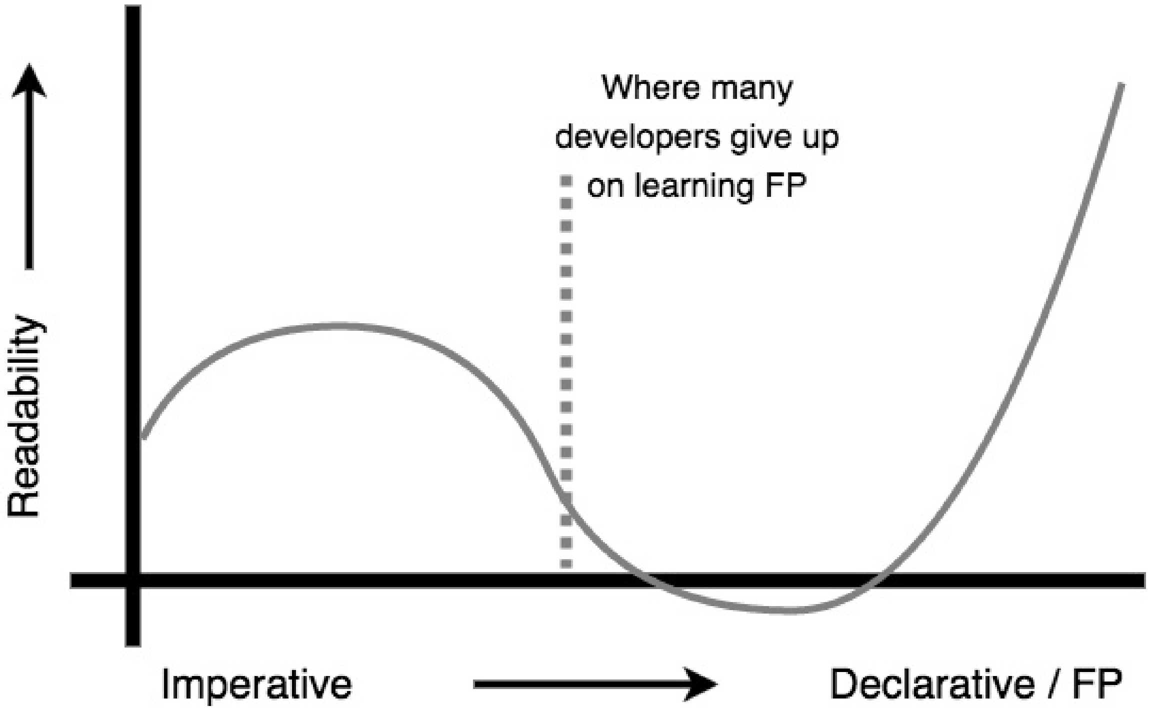
如果我们使用函数式编程原则，我相信我们将写出更容易理解的代码。一旦我们知道这些原则，它们将在代码中被识别和熟悉，这意味着当我们读取一段代码时，我们将花费更少的时间来进行定位。我们的重点将在于如何组建所有已知的“乐高片段”，而不是这些“乐高片段”是什么意思。

函数式编程是编写可读代码的最有效工具之一（可能还有其他）。这就是为什么函数式编程如此重要。

可读性曲线

很重要的是，我先花点时间来讲述一种多年来让我感到困惑和沮丧的现象，在写本书时该问题尤为尖锐。

这也可能是许多开发人员会遇到的问题。亲爱的读者，当你读这篇文章的时候，你可能会发现自己也会遇到同样的状况。但是要振作起来，坚持下去，陡峭的学习曲线总会过去。



我们将在下一章更深入的讨论这个问题。但是你可能写过一些命令式的代码，像 `if` 语句和 `for` 循环这样的语句。这些语句旨在精确地指导计算机如何完成一件事情。声明式代码，以及我们努力遵循函数式编程原则所写出的代码，更专注于描述最终的结果。

还有个残酷的问题摆在眼前，我在写本书时花费了很多时间在此问题上：我需要花费更多的精力和编写更多的代码来提高代码的可读性，尽量减少乃至消除可能会引入程序错误的代码部分。

如果你期望用函数式编程重构过的代码能够立刻变得更美观、优雅、智能和简洁的话，这个有点不太现实，这个变化是需要一个过程的。

函数式编程以另一种方式来思考代码应该如何组织才能使数据流更加明显，并能让读者很快理解你的思想。这种努力是非常值得的，然而过程很艰辛，你可能需要花很多时间基于函数式编程来调整代码直到代码可读性变得好一些。

另外，我的经验是，转换为声明式的代码之前，大约需要做六次尝试。对我来说，编写符合函数式编程的代码更像是一个过程，而不是从一个范例到另一个范例的二进制转换。

我也会经常对写过的代码进行重构。就是说，写完一段代码，过几个小时或一天再看会有不一样的感觉。通常，重构之前的代码是比较混乱不堪，所以需要反复调整。

函数式编程的过程并没有让我在艺术的画布上笔下生辉，让观众拍案叫好。相反，编程的过程很艰辛且历历在目，感觉像坐在一辆不靠谱的马车穿过一片杂草丛生的灌木树林。

我并不是试图打消你的激情，而是真切希望你也能够在编程的道路上披荆斩棘。过后我终于看到可读性曲线向上延伸，所有付出都是值得的，我相信你也会有同样的感受。

接受

我们要系统的学习函数式编程，探索发现最基本的原则，我相信规范的函数式编程编程者会遵循这些原则并把它们作为开发的框架。但在大多数情况下，我们大都选择避开晦涩的术语或数学符号，否则很容易使学习者受挫。

我觉得一项技术你怎么称呼它不重要，重要的是理解它是什么并且它是怎么工作的。这并不是说共享术语不重要，它无疑可以简化经验丰富的专业人士之间的交流。但对学习者来说，它有点分散人的注意力。

所以我希望这本书能更多地关注基本概念而不是花哨的术语。这并不是说没有术语，肯定会有。但不要太沉迷于华丽的词藻，追寻其背后的含义，这正是本书的目的。

我把这种欠缺正式实践的编程思想称为“轻量级函数式编程”，因为我认为眞正的函数式编程的形式主义在于， 因为我认为如果你还不习惯函数式编程主张的思想，你可能很难用它。这不仅仅只是猜测，而是我的亲身经历。即使在传教函数式编程过程和完成这本书之后，我仍然可以说，函数式编程中术语和符号的形式化对于我来说是非常非常困难的。我已经再三尝试，发现大部分都是很难掌握的。

我知道很多函数式编程编程者会认为形式主义本身有助于学习。但我认为这是一个坑，当你试图用形式主义获得某种安慰时，你就会踩坑。但如果碰巧你有数学背景，甚至还有一些 CS 经验，这些问题对你来说就可能驾轻就熟。但是我们中的一些人不具备这些条件，不管我们怎么努力，形式主义总是阻碍我们前进。

因此，这本书介绍了一些我认为函数式编程会涉及到的概念，虽然不能直接让你受益但可以帮你逐步理解函数式编程整个过程。

你不需要它

如果你规划一个项目花了很长时间，那么别人一定会告诉你“YAGNI” —— “你不需要它”。这个原则主要来自极限编程，强调构建特性的高风险和成本，这个风险和成本源自于项目本身是否需要。

有时我们考虑到将来可能会用到一个功能，并且认为现在构建它能够使得构建其他应用时更容易，后来意识到我们猜错了，原来这个功能并不需要，或者需要的完全是另外一套。另外一种情形是我们预估的功能是正确的，但构建得太早的话，相当于占用了开发现有功能的时间。有点像赔了夫人又折兵。

YAGNI 挑战，告诉我们：即使有的功能在某种情况下是反直觉的，我们也常常应该推迟构建,直到当前需要这个功能。我们倾向于夸大一个功能未来重构成本的心理估计，但往往这个重构是在将来需要时才会做。

上述情况对函数式编程也同样适用，不过我还是要先敲个警钟: 本书包含了大量你想去尝试的有趣的开发模式，但这不意味着你的代码一定要使用这些模式。

我与很多函数式编程开发人员的不同之处在于：你掌握了函数式编程并不意味着你一定得用它。此外，解决问题的方法很多，即使你掌握了更精炼的方法，能对维护和可扩展性更"经得起未来的考验"，但更轻量的函数式编程模式可能更适合该场景。

一般来说，我建议你在代码中寻求平衡，并且当你掌握函数式编程的诀窍时，在应用的过程中也应保持谨慎。在决定某个模式或抽象概念是否能使得部分代码可读性提高，或是否只是引入更智能的库时，YAGNI 的原则同样适用。

> 提醒一句，一些未曾用过的扩展点不仅浪费精力，而且可能妨碍你的工作。
>
> Jeremy D. Miller @jeremydmiller 2/20/15
>
> https://twitter.com/jeremydmiller/status/568797862441586688

记住，你编写的每一行代码之后都要有人来维护，这个人可能是你的团队成员，也可能是未来的你。如果代码写的太过复杂，那么无论谁来维护都会对你炫技式的故作聪明的做法倍感压力。

最好的代码是可读性高的代码，因为它在正确的（理想主义）和必然的（正确的）之间寻求到了恰到好处的平衡。

资源

我撰写这篇文章的过程参考了许多不同的资源。我相信你也会从中受益，所以我想花点时间把它们列出来。

书籍推荐

一些你务必要阅读的函数式编程 / JavaScript 书籍：

- Professor Frisby's Mostly Adequate Guide to Functional Programming by Brian Lonsdorf
- JavaScript Allongé by Reg Braithwaite
- Functional JavaScript by Michael Fogus

博客和站点

一些其他作者和相关内容供查阅：

- Fun Fun Function Videos by Mattias P Johansson
- Awesome函数式编程JS

- Kris Jenkins
- Eric Elliott
- James A Forbes
- James Longster
- André Staltz
- Functional Programming Jargon
- Functional Programming Exercises

一些库

本书中的代码段不使用库。我们发现的每一个操作，将派生出如何在独立的、普通的 JavaScript 中实现它。然而，当你开始使用函数式编程构建更多的眞正代码时，你很快就会使用现有库中所提供的更可靠高效的通用功能。

顺便说一下，你要确保检查你所使用的库函数的文档，以确保你知道它们是如何工作的。它与本文中构建的代码有许多相似之处，但毫无疑问即便跟最流行的库相比还是会存在一些差异。

下面是一些流行的 JavaScript 版本的函数式编程库，可以开启你的探索之路：

- Ramda
- lodash/fp
- functional.js
- Immutable.js

附录 C 展示了用到了本书中一些示例的库。

总结

这就是 JavaScript 轻量级函数式编程。我们的目标是学会与代码交流，而不是在符号或术语的大山下被压的喘不过气。希望这本书能开启你的旅程！

JavaScript 轻量级函数式编程

第 2 章：函数基础

函数式编程不是仅仅用 `function` 这个关键词来编程。如果真这么简单，那我这本书可以到此为止了！重点在于：函数是函数式编程的核心。这也是如何使用函数（function）才能使我们的代码具有函数式（functional）的方法。

然而，你真的明白函数的含义吗？

在这一章，我们将会介绍函数的基础知识，为阅读本书的后续章节打下基础。从某些方面来讲，这章回顾的函数知识并不是针对函数式编程者，非函数式编程者同样需要了解。但如果我们想要充分、全面地学习函数式编程的概念，我们需要从里到外地理解函数。

请做好准备，因为还有好多你未知的函数知识。

什么是函数？

针对函数式编程，很自然而然的我会想到从函数开始。这太明显不过了，但是我认为我们需要扎实地走好旅程的第一步。

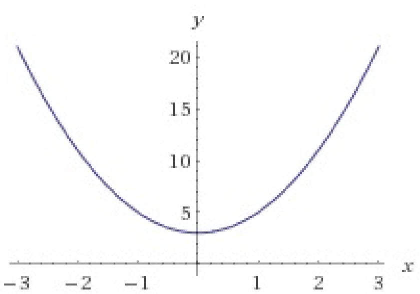
所以......什么是函数？

简要的数学回顾

我知道我曾说过，离数学越远越好，但是让我们暂且忍一小段时间，在这段时间里，我们会尽快地回顾在代数中一些函数和图像的基本知识。

你还记得你在学校里学习任何有关 `f(x)` 的知识吗？还有方程 `y = f(x)` ？

现有方程式定义如下：`f(x) = 2x2 + 3` 。这个方程有什么意义？它对应的图像是什么样的呢？如下图：



你可以注意到：对于 `x` 取任意值，例如 `2`，带入方程后会得到 `11`。这里的 `11` 代表函数的返回值，更简单来说就是 `y` 值。

根据上述，现在有一个点 `(2,11)` 在图像的曲线上，并且当我们有一个 `x` 值，我们都能获得一个对应的 `y` 值。把两个值组合就能得到一个点的坐标，例如 `(0,3)`，`(-1,5)`。当把所有的这些点放在一起，就会获得这个抛物线方程的图像，如上图所示。

所以，这些和函数式编程有什么关系？

在数学中，函数总是获取一些输入值，然后给出一个输出值。你能听到一个函数式编程的术语叫做“态射”：这是一个优雅的方式来描述一组值和另一组值的映射关系，就像一个函数的输入值与输出值之间的关联关系。

在代数数学中，那些输入值和输出值经常代表着绘制坐标的一部分。不过，在我们的程序中，我们可以定义函数有各种的输入和输出值，并且它们不需要和绘制在图表上的曲线有任何关系。

函数 vs 程序

为什么所有的讨论都围绕数学和图像？因为在某种程度上，函数式编程就是使用在数学意义上的方程作为函数。

你可能会习以为常地认为函数就是程序。它们之间的区别是什么？程序就是一个任意的功能集合。它或许有许多个输入值，或许没有。它或许有一个输出值（ `return` 值），或许没有。

而函数则是接收输入值，并明确地 `return` 值。

如果你计划使用函数式编程，你应该尽可能多地使用函数，而不是程序。你所有编写的 `function` 应该接收输入值，并且返回输出值。这么做的原因是多方面的，我们将会在后面的书中来介绍的。

函数输入

从上述的定义出发，所有的函数都需要输入。

你有时听人们把函数的输入值称为 “arguments” 或者 “parameters” 。所以它到底是什么？

arguments 是你输入的值（实参）， *parameters* 是函数中的命名变量（形参），用于接收函数的输入值。例子如下：

```
function foo(x,y) {
    // ..
}

var a = 3;

foo( a, a * 2 );
```

`a` 和 `a * 2`（即为 `6`）是函数 `foo(..)` 调用的 *arguments*。`x` 和 `y` 是 *parameters*，用于接收参数值（分别为 `3` 和 `6`）。

注意： 在 JavaScript 中，实参的个数没必要完全符合形参的个数。如果你传入许多个实参，而且多过你所声明的形参，这些值仍然会原封不动地被传入。你可以通过不同的方式去访问，包含了你以前可能听过的老办法 —— `arguments` 对象。反之，你传入少于声明形参个数的实参，所有缺少的参数将会被赋予 `undefined` 变量，意味着你仍然可以在函数作用域中使用它，但值是 `undefined`。

输入计数

一个函数所“期望”的实参个数是取决于已声明的形参个数，即你希望传入多少参数。

```
function foo(x,y,z) {
    // ..
}
```

`foo(..)` 期望三个实参，因为它声明了三个形参。这里有一个特殊的术语：Arity。Arity 指的是一个函数声明的形参数量。 `foo(..)` 的 Arity 是 `3`。

你可能需要在程序运行时获取函数的 Arity，使用函数的 `length` 属性即可。

```
function foo(x,y,z) {
    // ..
}

foo.length;             // 3
```

在执行时要确定 Arity 的一个原因是：一段代码接受一个函数的指针引用，有可能这个引用指向不同来源，我们要根据这些来源的 Arity 传入不同的参数值。

举个例子，如果 `fn` 可能指向的函数分别期望 1、2 或 3 个参数，但你只希望把变量 `x` 放在最后的位置传入：

```
// fn 是一些函数的引用
// x 是存在的值

if (fn.length == 1) {
    fn( x );
}
else if (fn.length == 2) {
    fn( undefined, x );
}
else if (fn.length == 3) {
    fn( undefined, undefined, x );
}
```

提示： 函数的 `length` 属性是一个只读属性，并且它是在最初声明函数的时候就被确定了。它应该当做用来描述如何使用该函数的一个基本元数据。

需要注意的是，某些参数列表的变量会让 `length` 属性变得不同于你的预期。别紧张，我们将会在后续的章节逐一解释这些特性（引入 ES6）：

```
function foo(x,y = 2) {
    // ..
}

function bar(x,...args) {
    // ..
}

function baz( {a,b} ) {
    // ..
}

foo.length;             // 1
bar.length;             // 1
baz.length;             // 1
```

如果你使用这些形式的参数，你或许会被函数的 `length` 值吓一跳。

那我们怎么得到当前函数调用时所接收到的实参个数呢？这在以前非常简单，但现在情况稍微复杂了一些。每一个函数都有一个 `arguments` 对象（类数组）存放需要传入的参数。你可以通过 `arguments` 的 `length` 值来找出有多少传入的参数：

```
function foo(x,y,z) {
    console.log( arguments.length );    // 2
}

foo( 3, 4 );
```

由于 ES5（特别是严格模式下）的 `arguments` 不被一些人认同，很多人尽可能地避免使用。尽管如此，它永远不会被移除，这是因为在 JS 中我们“永远不会”因为便利性而去牺牲向后的兼容性，但我还是强烈建议不要去使用它。

然而，当你需要知道参数个数的时候，`arguments.length` 还是可以用的。在未来版本的 JS 或许会新增特性来替代 `arguments.length`，如果成真，那么我们可以完全把 `arguments` 抛诸脑后。

请注意：不要通过 `arguments[1]` 访问参数的位置。只要记住 `arguments.length`。

除此之外，你或许想知道如何访问那些超出声明的参数？这个问题我一会儿会告诉你，不过你先要问自己的问题是，“为什么我想要知道这个？”。认真地思考一段时间。

发生这种情况应该是非常罕见的。因为这不会是你日常需要的，也不会是你编写函数时所必要的东西。如果这种情况真的发生，你应该花 20 分钟来试着重新设计函数，或者命名那些多出来的参数。

带有可变数量参数的函数被称为 variadic。有些人更喜欢这样的函数设计，不过你会发现，这正是函数式编程者想要避免的。

好了，上面的重点已经讲得够多了。

例如，当你需要像数组那样访问参数，很有可能的原因是你想要获取的参数没有在一个规范的位置。我们如何处理？

ES6 救星来了！让我们用 `...` 操作符声明我们的函数，也被当做 “spread”、“rest” 或者 “gather” (我比较偏爱)提及。

```
function foo(x,y,z,...args) {
    // ..
}
```

看到参数列表中的 `...args` 了吗？那就是 ES6 用来告诉解析引擎获取所有剩余的未命名参数，并把它们放在一个真实的命名为 `args` 的数组。`args` 无论是不是空的，它永远是一个数组。但它不包含已经命名的 `x`，`y` 和 `z` 参数，只会包含超出前三个值的传入参数。

```
function foo(x,y,z,...args) {
    console.log( x, y, z, args );
}

foo();                          // undefined undefined undefined []
foo( 1, 2, 3 );                 // 1 2 3 []
foo( 1, 2, 3, 4 );              // 1 2 3 [ 4 ]
foo( 1, 2, 3, 4, 5 );           // 1 2 3 [ 4, 5 ]
```

所以，如果你诚心想要设计一个函数，并且计算出任意传入参数的个数，那就在最后用 `...args` （或任何你喜欢的名称）。现在你有一个真正的、好用的数组来获取这些参数值了。

你需要注意的是： `4` 所在的位置是 `args` 的第 `0` 个，不是在第 `3` 个位置。它的 `length` 值也不包含 `1` 、 `2` 和 `3` ， `...args` 剩下所有的值, 但不包括 `x` 、 `y` 和 `z` 。

你甚至可以直接在参数列中使用 `...` 操作符，没有其他正式声明的参数也没关系：

```
function foo(...args) {
    // ..
}
```

现在 `args` 是一个由参数组成的完整数组，你可以尽情使用 `args.length` 来获取传入的参数。你也可以安全地使用 `args[1]` 或者 `args[317]` 。当然，别真的传 318 个参数！

说到 ES6 的好，你肯定想知道一些小秘诀。在这里将会介绍一些，更多的内容推荐你阅读《You Don't Know JS: ES6 & Beyond》这本书的第 2 章。

关于实参的小技巧

如果你希望调用函数的时候只传一个数组代替之前的多个参数，该怎么办？

```
function foo(...args) {
    console.log( args[3] );
}

var arr = [ 1, 2, 3, 4, 5 ];

foo( ...arr );                          // 4
```

我们的新朋友 `...` 在这里被使用到了，但不仅仅在形参列表，在函数调用的时候，同样使用在实参列表。在这里的情况有所不同：在形参列表，它把实参整合。在实参列表，它把实参展开。所以 `arr` 的内容是以函数 `foo(..)` 引用的单独参数进行展开。你能理解传入一个引用值和传入整个 `arr` 数组两者之间的不同了吗？

顺带一提，多个值和 `...` 是可以相互交错放置的，如下：

```
var arr = [ 2 ];

foo( 1, ...arr, 3, ...[4,5] );          // 4
```

在对称的意义上来考虑 `...` ：在值列表的情况，它会展开。在赋值的情况，它就像形参列表一样，因为实参会赋值到形参上。

无论采取什么行为，`...` 都会让实参数组更容易操作。那些我们使用实参数组 `slice(..)`，`concat(..)` 和 `apply(..)` 的日子已经过去了。

关于形参的小技巧

在 ES6 中，形参可以声明默认值。当形参没有传入到实参中，或者传入值是 `undefined`，会进行默认赋值的操作。

思考下面代码：

```
function foo(x = 3) {
    console.log( x );
}

foo();                    // 3
foo( undefined );         // 3
foo( null );              // null
foo( 0 );                 // 0
```

注意：我们不会更加详细地解释了，但是默认值表达式是惰性的，这意味着仅当需要的时候，它才会被计算。它同样也可以是一些有效的 JS 表达式，甚至一个函数引用。许多非常酷的小技巧用到了这个方法。例如，你可以这样在你的参数列声明 `x = required()`，并且在函数 `required()` 中 `抛出 "This argument is required."` 来确信总有人用你指定的实参或形参来引用你的函数。

另一个我们可以在参数中使用的 ES6 技巧，被称为“解构”。在这里我们只会简单一提，因为要说清这个话题实在太过繁杂。在这里推荐《ES6 & Beyond》这本书了解更多信息。

还记得我们之前提到的可以接受 318 个参数的 `foo(..)` 吗？

```
function foo(...args) {
    // ..
}

foo( ...[1,2,3] );
```

如果我们想要把函数内的参数从一个个单独的参数值替换为一个数组，应该怎么做？这里有两个 `...` 的写法：

```
function foo(args) {
    // ..
}

foo( [1,2,3] );
```

这个非常简单。但如果我们想要命名传入数组的第 1、2 个值，该怎么做？我们不能用单独传入参数的办法了，所以这似乎看起来无能为力。不过解构可以回答这个问题：

```
function foo( [x,y,...args] = [] ) {
    // ..
}

foo( [1,2,3] );
```

你看到了在参数列出现的 `[ .. ]` 了吗？这就是数组解构。解构是通过你期望的模式来描述数据（对象，数组等），并分配（赋值）值的一种方式。

在这里例子中，解构告诉解析器，一个数组应该出现的赋值位置（即参数）。这种模式是：拿出数组中的第一个值，并且赋值给局部参数变量 `x`，第二个赋值给 `y`，剩下的则组成 `args`。

你可以通过自己手动处理达到同样的效果：

```
function foo(params) {
    var x = params[0];
    var y = params[1];
    var args = params.slice( 2 );

    // ..
}
```

现在我们可以发现，在我们这本书中要多次提到的第一条原则：声明性代码通常比命令式代码更干净。

声明式代码，如同之前代码片段里的解构，强调一段代码的输出结果。命令式代码，像刚才我们自己手动赋值的例子，注重的是如何得到结果。如果你稍晚再读这一段代码，你必须在脑子里面再执行一遍才能得到你想要的结果。这个结果是编写在这儿，但是不是直接可见的。

只要可能，无论我们的语言和我们的库或框架允许我们达到什么程度，我们都应该尽可能使用声明性的和自解释的代码。

正如我们可以解构的数组，我们可以解构的对象参数：

```
function foo( {x,y} = {} ) {
    console.log( x, y );
}

foo( {
    y: 3
} );                    // undefined 3
```

我们传入一个对象作为一个参数，它解构成两个独立的参数变量 `x` 和 `y`，从传入的对象中分配相应属性名的值。我们不在意属性值 `x` 到底存不存在对象上，如果不存在，它最终会如你所想被赋值为 `undefined` 。

但是我希望你注意：对象解构的部分参数是将要传入 `foo(..)` 的对象。

现在有一个正常可用的调用现场 `foo(undefined,3)`，它用于映射实参到形参。我们试着把 `3` 放到第二个位置，分配给 `y` 。但是在新的调用现场上用到了参数解构，一个简单的对象属性代表了实参 `3` 应该分配给形参(`y`)。

我们不需要操心 `x` 应该放在哪个调用现场。因为事实上，我们不用去关心 `x`，我们只需要省略它，而不是分配 `undefined` 值。

有一些语言对这样的操作有一个直接的特性：命名参数。换句话说，在调用现场，通过标记输入值来告诉它映射关系。JavaScript 没有命名参数，不过退而求其次，参数对象解构是一个选择。

使用对象解构来传入多个匿名参数是函数式编程的优势，这个优势在于使用一个参数（对象）的函数能更容易接受另一个函数的单个输出。这点会在后面讨论到。

回想一下，术语 Arity 是指期望函数接收多少个参数。Arity 为 1 的函数也被称为一元函数。在函数式编程中，我们希望我们的函数在任何的情况下是一元的，有时我们甚至会使用各种技巧来将高 Arity 的函数都转换为一元的形式。

注意： 在第 3 章，我们将重新讨论命名参数的解构技巧，并使用它来处理关于参数排序的问题。

随着输入而变化的函数

思考以下函数

```
function foo(x,y) {
    if (typeof x == "number" && typeof y == "number") {
        return x * y;
    }
    else {
        return x + y;
    }
}
```

明显地，这个函数会根据你传入的值而有所不同。

举例：

```
foo( 3, 4 );            // 12

foo( "3", 4 );            // "34"
```

程序员这样定义函数的原因之一是，更容易通过同一个函数来重载不同的功能。最广为人知的例子就是 jQuery 提供的 `$(..)`。"$" 函数大约有十几种不同的功能 —— 从 DOM 元素查找，到 DOM 元素创建，到等待 “DOMContentLoaded” 事件后，执行一个函数，这些都取决于你传递给它的参数。

上述函数，显而易见的优势是 API 变少了（仅仅是一个 `$(..)` 函数），但缺点体现在阅读代码上，你必须仔细检查传递的内容，理解一个函数调用将做什么。

通过不同的输入值让一个函数重载拥有不同的行为的技巧叫做特定多态（ad hoc polymorphism）。

这种设计模式的另一个表现形式就是在不同的情况下，使函数具有不同的输出（在下一章节会提到）。

警告： 要对方便的诱惑有警惕之心。因为你可以通过这种方式设计一个函数，即使可以立即使用，但这个设计的长期成本可能会让你后悔。

函数输出

在 JavaScript 中，函数只会返回一个值。下面的三个函数都有相同的 `return` 操作。

```
function foo() {}

function bar() {
    return;
}

function baz() {
    return undefined;
}
```

如果你没有 `return` 值，或者你使用 `return;`，那么则会隐式地返回 `undefined` 值。

如果想要尽可能靠近函数式编程的定义：使用函数而非程序，那么我们的函数必须永远有返回值。这也意味着他们必须明确地 `return` 一个值，通常这个值也不是 `undefined`。

一个 `return` 的表达式仅能够返回一个值。所以，如果你需要返回多个值，切实可行的办法就是把你需要返回的值放到一个复合值当中去，例如数组、对象：

```
function foo() {
    var retValue1 = 11;
    var retValue2 = 31;
    return [ retValue1, retValue2 ];
}
```

解构方法可以使用于解构对象或者数组类型的参数，也可以使用在平时的赋值当中：

```
function foo() {
    var retValue1 = 11;
    var retValue2 = 31;
    return [ retValue1, retValue2 ];
}

var [ x, y ] = foo();
console.log( x + y );           // 42
```

将多个值集合成一个数组（或对象）做为返回值，然后再解构回不同的值，这无形中让一个函数能有多个输出结果。

提示： 在这里我十分建议你花一点时间来思考：是否需要避免函数有可重构的多个输出？或许将这个函数分为两个或更多个更小的单用途函数。有时会需要这么做，有时可能不需要，但你应该至少考虑一下。

提前 return

`return` 语句不仅仅是从函数中返回一个值，它也是一个流量控制结构，它可以结束函数的执行。因此，具有多个 `return` 语句的函数具有多个可能的退出点，这意味着如果输出的路径很多，可能难以读取并理解函数的输出行为。

思考以下：

```
function foo(x) {
    if (x > 10) return x + 1;

    var y = x / 2;

    if (y > 3) {
        if (x % 2 == 0) return x;
    }

    if (y > 1) return y;

    return x;
}
```

突击测验：不要作弊也不要在浏览器中运行这段代码，请思考 `foo(2)` 返回什么？ `foo(4)` 返回什么？ `foo(8)` ， `foo(12)` 呢？

你对自己的回答有多少信心？你付出多少精力来获得答案？我错了两次后，我试图仔细思考并且写下来！

我认为在许多可读性的问题上，是因为我们不仅使用 `return` 返回不同的值，更把它作为一个流控制结构——在某些情况下可以提前退出一个函数的执行。我们显然有更好的方法来编写流控制（ `if` 逻辑等），也有办法使输出路径更加明显。

注意： 突击测验的答案是： `2` ， `2` ， `8` 和 `13` 。

思考以下版本的代码：

```
function foo(x) {
    var retValue;

    if (retValue == undefined && x > 10) {
        retValue = x + 1;
    }

    var y = x / 2;

    if (y > 3) {
        if (retValue == undefined && x % 2 == 0) {
            retValue = x;
        }
    }

    if (retValue == undefined && y > 1) {
        retValue = y;
    }

    if (retValue == undefined) {
        retValue = x;
    }

    return retValue;
}
```

这个版本毫无疑问是更冗长的。但是在逻辑上，我认为这比上面的代码更容易理解。因为在每个 `retValue` 可以被设置的分支， 这里都有个守护者以确保 `retValue` 没有被设置过才执行。

相比在函数中提早使用 `return` ，我们更应该用常用的流控制（ `if` 逻辑 ）来控制 `retValue` 的赋值。到最后，我们 `return retValue` 。

我不是说，你只能有一个 `return` ，或你不应该提早 `return` ，我只是认为在定义函数时，最好不要用 `return` 来实现流控制，这样会创造更多的隐含意义。尝试找出最明确的表达逻辑的方式，这往往是最好的办法。

未 `return` 的输出

有个技巧你可能在你的大多数代码里面使用过，并且有可能你自己并没有特别意识到，那就是让一个函数通过改变函数体外的变量产出一些值。

还记得我们之前提到的函数 `f(x) = 2x2 + 3` 吗？我们可以在 JS 中这样定义：

```
var y;

function foo(x) {
    y = (2 * Math.pow( x, 2 )) + 3;
}

foo( 2 );

y;                          // 11
```

我知道这是一个无聊的例子。我们完全可以用 `return` 来返回，而不是赋值给 `y` ：

```
function foo(x) {
    return (2 * Math.pow( x, 2 )) + 3;
}

var y = foo( 2 );

y;                          // 11
```

这两个函数完成相同的任务。我们有什么理由要从中挑一个吗？是的，绝对有。

解释这两者不同的一种方法是，后一个版本中的 `return` 表示一个显式输出，而前者的 `y` 赋值是一个隐式输出。在这种情况下，你可能已经猜到了：通常，开发人员喜欢显式模式而不是隐式模式。

但是，改变一个外部作用域的变量，就像我们在 `foo(..)` 中所做的赋值 `y` 一样，只是实现隐式输出的一种方式。一个更微妙的例子是通过引用对非局部值进行更改。

思考：

```
function sum(list) {
    var total = 0;
    for (let i = 0; i < list.length; i++) {
        if (!list[i]) list[i] = 0;

        total = total + list[i];
    }

    return total;
}

var nums = [ 1, 3, 9, 27, , 84 ];

sum( nums );            // 124
```

很明显，这个函数输出为 `124`，我们也非常明确地 `return` 了。但你是否发现其他的输出？查看代码，并检查 `nums` 数组。你发现区别了吗？

为了填补 `4` 位置的空值 `undefined`，这里使用了 `0` 代替。尽管我们在局部操作 `list` 参数变量，但我们仍然影响了外部的数组。

为什么？因为 `list` 使用了 `nums` 的引用，不是对 `[1,3,9,..]` 的值复制，而是引用复制。因为 JS 对数组、对象和函数都使用引用和引用复制，我们可以很容易地从函数中创建输出，即使是无心的。

这个隐式函数输出在函数式编程中有一个特殊的名称：副作用。当然，没有副作用的函数也有一个特殊的名称：纯函数。我们将在以后的章节讨论这些，但关键是我们应该喜欢纯函数，并且要尽可能地避免副作用。

函数功能

函数是可以接受并且返回任何类型的值。一个函数如果可以接受或返回一个甚至多个函数，它被叫做高阶函数。

思考：

```
function forEach(list,fn) {
    for (let i = 0; i < list.length; i++) {
        fn( list[i] );
    }
}

forEach( [1,2,3,4,5], function each(val){
    console.log( val );
} );
// 1 2 3 4 5
```

`forEach(..)` 就是一个高阶函数，因为它可以接受一个函数作为参数。

一个高阶函数同样可以把一个函数作为输出，像这样：

```
function foo() {
    var fn = function inner(msg){
        console.log( msg );
    };

    return fn;
}

var f = foo();

f( "Hello!" );            // Hello!
```

`return` 不是“输出”函数的唯一办法。

```
function foo() {
    var fn = function inner(msg){
        console.log( msg );
    };

    bar( fn );
}

function bar(func) {
    func( "Hello!" );
}

foo();                    // Hello!
```

将其他函数视为值的函数是高阶函数的定义。函数式编程者们应该学会这样写！

保持作用域

在所有编程，尤其是函数式编程中，最强大的就是：当一个函数内部存在另一个函数的作用域时，对当前函数进行操作。当内部函数从外部函数引用变量，这被称作闭包。

实际上，闭包是它可以记录并且访问它作用域外的变量，甚至当这个函数在不同的作用域被执行。

思考：

```
function foo(msg) {
    var fn = function inner(){
        console.log( msg );
    };

    return fn;
}

var helloFn = foo( "Hello!" );

helloFn();                // Hello!
```

处于 `foo(..)` 函数作用域中的 `msg` 参数变量是可以在内部函数中被引用的。当 `foo(..)` 执行时，并且内部函数被创建，函数可以获取 `msg` 变量，即使 `return` 后仍可被访问。

虽然我们有函数内部引用 `helloFn`，现在 `foo(..)` 执行后，作用域应该回收，这也意味着 `msg` 也不存在了。不过这个情况并不会发生，函数内部会因为闭包的关系，将 `msg` 保留下来。只要内部函数（现在被处在不同作用域的 `helloFn` 引用）存在，`msg` 就会一直被保留。

让我们看看闭包作用的一些例子：

```
function person(id) {
    var randNumber = Math.random();

    return function identify(){
        console.log( "I am " + id + ": " + randNumber );
    };
}

var fred = person( "Fred" );
var susan = person( "Susan" );

fred();                   // I am Fred: 0.8331252801601532
susan();                  // I am Susan: 0.3940753308893741
```

`identify()` 函数内部有两个闭包变量，参数 `id` 和 `randNumber`。

闭包不仅限于获取变量的原始值：它不仅仅是快照，而是直接链接。你可以更新该值，并在下次访问时获取更新后的值。

```
function runningCounter(start) {
    var val = start;

    return function current(increment = 1){
        val = val + increment;
        return val;
    };
}

var score = runningCounter( 0 );

score();                // 1
score();                // 2
score( 13 );            // 15
```

警告： 我们将在之后的段落中介绍更多。不过在这个例子中，你需要尽可能避免使用闭包来记录状态更改（ `val` ）。

如果你需要设置两个输入，一个你已经知道，另一个还需要后面才能知道，你可以使用闭包来记录第一个输入值：

```
function makeAdder(x) {
    return function sum(y){
        return x + y;
    };
}

//我们已经分别知道作为第一个输入的 10 和 37
var addTo10 = makeAdder( 10 );
var addTo37 = makeAdder( 37 );

// 紧接着，我们指定第二个参数
addTo10( 3 );           // 13
addTo10( 90 );           // 100

addTo37( 13 );           // 50
```

通常， `sum(..)` 函数会一起接收 `x` 和 `y` 并相加。但是在这个例子中，我们接收并且首先记录（通过闭包） `x` 的值，然后等待 `y` 被指定。

注意： 在连续函数调用中指定输入，这种技巧在函数式编程中非常普遍，并且有两种形式：偏函数应用和柯里化。我们稍后会在文中深入讨论。

当然，因为函数如果只是 JS 中的值，我们可以通过闭包来记住函数值。

```
function formatter(formatFn) {
    return function inner(str){
        return formatFn( str );
    };
}

var lower = formatter( function formatting(v){
    return v.toLowerCase();
} );

var upperFirst = formatter( function formatting(v){
    return v[0].toUpperCase() + v.substr( 1 ).toLowerCase();
} );

lower( "WOW" );              // wow
upperFirst( "hello" );       // Hello
```

函数式编程并不是在我们的代码中分配或重复 `toUpperCase()` 和 `toLowerCase()` 逻辑，而是鼓励我们用优雅的封装方式来创建简单的函数。

具体来说，我们创建两个简单的一元函数 `lower(..)` 和 `upperFirst(..)`，因为这些函数在我们程序中，更容易与其他函数配合使用。

提示：你知道如何让 `upperFirst(..)` 使用 `lower(..)` 吗？

我们将在本书的后续中大量使用闭包。如果抛开整个编程来说，它可能是所有函数式编程中最重要的基础。希望你能用得舒服！

句法

在我们函数入门开始之前，让我们花点时间来讨论它的语法。

不同于本书中的许多其他部分，本节中的讨论主要是意见和偏好，无论你是否同意这里提出的观点或采取相反的观点。这些想法是非常主观的，尽管许多人似乎对此非常执着。不过最终，都由你决定。

什么是名称？

在语法上，函数声明需要包含一个名称：

```
function helloMyNameIs() {
    // ..
}
```

但是函数表达式可以命名或者匿名：

```
foo( function namedFunctionExpr(){
    // ..
} );

bar( function(){    // <-- 这就是匿名的!
    // ..
} );
```

顺便说一句，匿名的意思是什么？具体来说，函数具有一个 `name` 的属性，用于保存函数在语法上设定名称的字符串值，例如 `"helloMyNameIs"` 或 `"FunctionExpr"`。这个 `name` 属性特别用于 JS 环境的控制台或开发工具。当我们在堆栈轨迹中追踪（通常来自异常）时，这个属性可以列出该函数。

而匿名函数通常显示为：`(anonymous function)`。

如果你曾经试着在一个异常的堆栈轨迹中调试一个 JS 程序，你可能已经发现痛苦了：看到 `(anonymous function)` 出现。这个列表条目不给开发人员任何关于异常来源路径的线索。它没有给我们开发者提供任何帮助。

如果你命名了你的函数表达式，名称将会一直被使用。所以如果你使用了一个良好的名称 `handleProfileClicks` 来取代 `foo`，你将会在堆栈轨迹中获得更多的信息。

在 ES6 中，匿名表达式可以通过名称引用来获得名称。思考：

```
var x = function(){};

x.name;            // x
```

如果解析器能够猜到你可能希望函数采用什么名称，那么它将会继续下去。

但请注意，并不是所有的句法形式都可以用名称引用。最常见的地方是函数表达式是函数调用的参数：

```
function foo(fn) {
    console.log( fn.name );
}

var x = function(){};

foo( x );                // x
foo( function(){} );    //
```

当名称不能直接从周围的语法中被推断时，它仍会是一个空字符串。这样的函数将在堆栈轨迹中的被报告为一个 `(anonymous function)`。

除了调试问题之外，函数被命名还有一个其他好处。首先，句法名称（又称词汇名）是可以被函数内部的自引用。自引用是递归（同步和异步）所必需的，也有助于事件处理。

思考这些不同的情况：

```
// 同步情况:
function findPropIn(propName,obj) {
    if (obj == undefined || typeof obj != "object") return;

    if (propName in obj) {
        return obj[propName];
    }
    else {
        let props = Object.keys( obj );
        for (let i = 0; i < props.length; i++) {
            let ret = findPropIn( propName, obj[props[i]] );
            if (ret !== undefined) {
                return ret;
            }
        }
    }
}
```

```
// 异步情况:
setTimeout( function waitForIt(){
    // it 存在了吗?
    if (!o.it) {
        // 再试一次
        setTimeout( waitForIt, 100 );
    }
}, 100 );
```

```
// 事件处理未绑定
document.getElementById( "onceBtn" )
    .addEventListener( "click", function handleClick(evt){
        // 未绑定的 event
        evt.target.removeEventListener( "click", handleClick, false );

        // ..
    }, false );
```

在这些情况下，使用命名函数的函数名引用，是一种有用和可靠的在自身内部自引用的方式。

此外，即使在单行函数的简单情况下，命名它们往往会使代码更加明了，从而让以前没有阅读过的人更容易阅读：

```
people.map( function getPreferredName(person){
    return person.nicknames[0] || person.firstName;
} )
// ..
```

光看函数 `getPreferredName(..)` 的代码，并不能很明确告诉我们这里的操作是什么意图。但有名称就可以增加代码可读性。

经常使用匿名函数表达式的另一个地方是 IIFE （立即执行函数表达式）：

```
(function(){

    // 我是 IIFE!

})();
```

你几乎从没看到为 IIFE 函数来命名，但他们应该命名。为什么？我们刚刚提到过的原因：堆栈轨迹调试，可靠的自我引用和可读性。如果你想不出你的 IIFE 应该叫什么，请至少使用 IIFE：

```
(function IIFE(){

    // 现在你真的知道我叫 IIFE!

})();
```

我有许多个理由可以解释命名函数比匿名函数更可取。事实上，我甚至认为匿名函数都是不可取的。相比命名函数，他们没有任何优势。

写匿名功能非常容易，因为我们完全不用在想名称这件事上费神费力。

诚实来讲，我也像大家一样在这个地方犯错。我不喜欢在起名称这件事上浪费时间。我能想到命名一个函数的前 3 或 4 个名字通常是不好的。我必须反复思考这个命名。这个时候，我宁愿只是用一个匿名函数表达。

但是，我们把易写性拿来与易读性做交换，这不是一个好选择。因为懒而不想为你的函数命名，这是常见的使用匿名功能的借口。

命名所有单个函数。如果你对着你写的函数，想不出一个好名称，我明确告诉你，那是你并没有完全理解这个函数的目的——或者来说它的目的太广泛或太抽象。你需要重新设计功能，直到它更清楚。从这个角度说，一个名称会更明白清晰。

从我自己的经验中证明，在思考名称的过程中，我会更好地了解它，甚至重构其设计，以提高可读性和可维护性。这些时间的投入是值得的。

没有 `function` 的函数

到目前为止，我们一直在使用完整的规范语法功能。但是相信你也对新的 ES6 `=>` 箭头函数语法有所耳闻。

比较：

```
people.map( function getPreferredName(person){
    return person.nicknames[0] || person.firstName;
} )
// ..

people.map( person => person.nicknames[0] || person.firstName );
```

哇！

关键字 `function` 没了，`return`，`()` 括号，`{}` 花括号和 `;` 分号也是这样。所有这一切，都是我们与一个胖箭头做了交易： `=>` 。

但还有另一件事我们忽略了。 你发现了吗？`getPreferredName` 函数名也没了。

那就对了。 `=>` 箭头函数是词法匿名的。没有办法合理地为它提供一个名字。他们的名字可以像常规函数一样被推断，但是，最常见的函数表达式值作为参数的情况将不会起任何作用了。

假设 `person.nicknames` 因为一些原因没有被定义，一个异常将会被抛出，意味着这个 `(anonymous function)` 将会在追踪堆栈的最上层。啊！

`=>` 箭头函数的匿名性是 `=>` 的阿喀琉斯之踵。这让我不能遵守刚刚所说的命名原则了：阅读困难，调试困难，无法自我引用。

但是，这还不够糟糕，要面对的另一个问题是，如果你的函数定义有不同的场景，那么你必须要一大堆细微差别的语句来实现。我不会在这里详细介绍所有，但会简要地说：

```
people.map( person => person.nicknames[0] || person.firstName );

// 多个参数? 需要 ( )
people.map( (person,idx) => person.nicknames[0] || person.firstName );

// 解构参数? 需要 ( )
people.map( ({ person }) => person.nicknames[0] || person.firstName );

// 默认参数? 需要 ( )
people.map( (person = {}) => person.nicknames[0] || person.firstName );

// 返回对象? 需要 ( )
people.map( person =>
    ({ preferredName: person.nicknames[0] || person.firstName })
);
```

在函数式编程中， `=>` 令人兴奋的地方在于它几乎完全遵循函数的数学符号，特别是像 Haskell 这样的函数式编程语言。 `=>` 箭头函数语法甚至可以用于数学交流。

我们进一步地来深挖，我建议使用 `=>` 的论点是，通过使用更轻量级的语法，可以减少函数之间的视觉边界，也让我们使用偷懒的方式来使用它，这也是函数式编程者的另一个爱好。

我认为大多数的函数式编程者都会对此睁只眼闭只眼。他们喜欢匿名函数，喜欢简洁语法。但是像我之前说过的那样：这都由你决定。

注意： 虽然我不喜欢在我的应用程序中使用 `=>` ，但我们将在本书的其余部分多次使用它，特别是当我们介绍典型的函数式编程实战时，它能简化、优化代码片段中的空间。不过，增强或减弱代码的可读性也取决你自己做的决定。

来说说 This ？

如果您不熟悉 JavaScript 中的 `this` 绑定规则，我建议去看我写的《You Don't Know JS: this & Object Prototypes》。 出于这章的需要，我会假定你知道在一个函数调用（四种方式之一）中 `this` 是什么。但是如果你依然对 *this* 感到迷惑，告诉你个好消息，接下来我们会总结在函数式编程中你不应当使用 `this` 。

JavaScript 的 `function` 有一个 `this` 关键字，每个函数调用都会自动绑定。 `this` 关键字有许多不同的方式描述，但我更喜欢说它提供了一个对象上下文来使该函数运行。

`this` 是函数的一个隐式的输入参数。

思考：

```
function sum() {
    return this.x + this.y;
}

var context = {
    x: 1,
    y: 2
};

sum.call( context );        // 3

context.sum = sum;
context.sum();              // 3

var s = sum.bind( context );
s();                        // 3
```

当然，如果 `this` 能够隐式地输入到一个函数当中去，同样的，对象也可以作为显式参数传入：

```
function sum(ctx) {
    return ctx.x + ctx.y;
}

var context = {
    x: 1,
    y: 2
};

sum( context );
```

这样的代码更简单，在函数式编程中也更容易处理：当显性输入值时，我们很容易将多个函数组合在一起， 或者使用下一章输入适配技巧。然而当我们做同样的事使用隐性输入时，根据不同的场景，有时候会难处理，有时候甚至不可能做到。

还有一些技巧，是基于 `this` 完成的，例如原型授权（在《this & Object Prototypes》一书中也详细介绍）：

```
var Auth = {
    authorize() {
        var credentials = this.username + ":" + this.password;
        this.send( credentials, resp => {
            if (resp.error) this.displayError( resp.error );
            else this.displaySuccess();
        } );
    },
    send(/* .. */) {
        // ..
    }
};

var Login = Object.assign( Object.create( Auth ), {
    doLogin(user,pw) {
        this.username = user;
        this.password = pw;
        this.authorize();
    },
    displayError(err) {
        // ..
    },
    displaySuccess() {
        // ..
    }
} );

Login.doLogin( "fred", "123456" );
```

注意： `Object.assign(..)` 是一个 ES6+ 的实用工具，它用来将属性从一个或者多个源对象浅拷贝到目标对象： `Object.assign( target, source1, ... )` 。

这段代码的作用是：现在我们有两个独立的对象 `Login` 和 `Auth` ，其中 `Login` 执行原型授权给 `Auth` 。通过委托和隐式的 `this` 共享上下文对象，这两个对象在 `this.authorize()` 函数调用期间实际上是组合的，所以这个 `this` 上的属性或方法可以与 `Auth.authorize(..)` 动态共享 `this` 。

this 因为各种原因，不符合函数式编程的原则。其中一个明显的问题是隐式 `this` 共享。但我们可以更加显式地，更靠向函数式编程的方向：

```
// ..

authorize(ctx) {
    var credentials = ctx.username + ":" + ctx.password;
    Auth.send( credentials, function onResp(resp){
        if (resp.error) ctx.displayError( resp.error );
        else ctx.displaySuccess();
    } );
}

// ..

doLogin(user,pw) {
    Auth.authorize( {
        username: user,
        password: pw
    } );
}

// ..
```

从我的角度来看，问题不在于使用对象来进行操作，而是我们试图使用隐式输入取代显式输入。当我戴上名为函数式编程的帽子时，我应该把 `this` 放回衣架上。

总结

函数是强大的。

现在，让我们清楚地理解什么是函数：它不仅仅是一个语句或者操作的集合，而且需要一个或多个输入（理想情况下只需一个！）和一个输出。

函数内部的函数可以取到闭包外部变量，并记住它们以备日后使用。这是所有程序设计中最重要的概念之一，也是函数式编程的基础。

要警惕匿名函数，特别是 `=>` 箭头函数。虽然在编程时用起来很方便，但是会对增加代码阅读的负担。我们学习函数式编程的全部理由是为了书写更具可读性的代码，所以不要赶时髦去用匿名函数。

别用 `this` 敏感的函数。这不需要理由。

JavaScript 轻量级函数式编程

第 3 章：管理函数的输入（Inputs）

在第 2 章的“函数输入”小节中，我们聊到了函数形参（parameters）和实参（arguments）的基本知识，实际上还了解到一些能简化其使用方式的语法技巧，比如 `...` 操作符和解构（destructuring）。

在那个讨论中，我建议尽可能设计单一形参的函数。但实际上你不能每次都做到，而且也不能每次都掌控你的函数签名（译者注：JS 中，函数签名一般包含函数名和形参等函数关键信息，例如 `foo(a, b = 1, c)`）。

现在，我们把注意力放在更复杂、强大的模式上，以便讨论处在这些场景下的函数输入。

立即传参和稍后传参

如果一个函数接收多个实参，你可能会想先指定部分实参，余下的稍后再指定。

来看这个函数：

```
function ajax(url,data,callback) {
    // ..
}
```

想象一个场景，你要发起多个已知 URL 的 API 请求，但这些请求的数据和处理响应信息的回调函数要稍后才能知道。

当然，你可以等到这些东西都确定后再发起 `ajax(..)` 请求，并且到那时再引用全局 URL 常量。但我们还有另一种选择，就是创建一个已经预设 `url` 实参的函数引用。

我们将创建一个新函数，其内部仍然发起 `ajax(..)` 请求，此外在等待接收另外两个实参的同时，我们手动将 `ajax(..)` 第一个实参设置成你关心的 API 地址。

```
function getPerson(data,cb) {
    ajax( "http://some.api/person", data, cb );
}

function getOrder(data,cb) {
    ajax( "http://some.api/order", data, cb );
}
```

手动指定这些外层函数当然是完全有可能的，但这可能会变得冗长乏味，特别是不同的预设实参还会变化的时候，譬如：

```
function getCurrentUser(cb) {
    getPerson( { user: CURRENT_USER_ID }, cb );
}
```

函数式编程者习惯于在重复做同一种事情的地方找到模式，并试着将这些行为转换为逻辑可重用的实用函数。实际上，该行为肯定已是大多数读者的本能反应了，所以这并非函数式编程独有。但是，对函数式编程而言，这个行为的重要性是毋庸置疑的。

为了构思这个用于实参预设的实用函数，我们不仅要着眼于之前提到的手动实现方式，还要在概念上审视一下到底发生了什么。

用一句话来说明发生的事情：`getOrder(data,cb)` 是 `ajax(url,data,cb)` 函数的偏函数（**partially-applied functions**）。该术语代表的概念是：在函数调用现场（function call-site），将实参应用（**apply**） 于形参。如你所见，我们一开始仅应用了部分实参 —— 具体是将实参应用到 `url` 形参 —— 剩下的实参稍后再应用。

关于该模式更正式的说法是：偏函数严格来讲是一个减少函数参数个数（arity）的过程；这里的参数个数指的是希望传入的形参的数量。我们通过 `getOrder(..)` 把原函数 `ajax(..)` 的参数个数从 3 个减少到了 2 个。

让我们定义一个 `partial(..)` 实用函数：

```
function partial(fn,...presetArgs) {
    return function partiallyApplied(...laterArgs){
        return fn( ...presetArgs, ...laterArgs );
    };
}
```

建议： 只是走马观花是不行的。请花些时间研究一下该实用函数中发生的事情。请确保你真的理解了。由于在接下来的文章里，我们将会一次又一次地提到该模式，所以你最好现在就适应它。

`partial(..)` 函数接收 `fn` 参数，来表示被我们偏应用实参（partially apply）的函数。接着，`fn` 形参之后，`presetArgs` 数组收集了后面传入的实参，保存起来稍后使用。

我们创建并 `return` 了一个新的内部函数（为了清晰明了，我们把它命名为 `partiallyApplied(..)`），该函数中，`laterArgs` 数组收集了全部实参。

你注意到在内部函数中的 `fn` 和 `presetArgs` 引用了吗？他们是怎么如何工作的？在函数 `partial(..)` 结束运行后，内部函数为何还能访问 `fn` 和 `presetArgs` 引用？你答对了，就是因为闭包！内部函数 `partiallyApplied(..)` 封闭（closes over）了 `fn` 和 `presetArgs` 变

量，所以无论该函数在哪里运行，在 `partial(..)` 函数运行后我们仍然可以访问这些变量。所以理解闭包是多么的重要！

当 `partiallyApplied(..)` 函数稍后在某处执行时，该函数使用被闭包作用（closed over）的 `fn` 引用来执行原函数，首先传入（被闭包作用的）`presetArgs` 数组中所有的偏应用（partial application）实参，然后再进一步传入 `laterArgs` 数组中的实参。

如果你对以上感到任何疑惑，请停下来再看一遍。相信我，随着我们进一步深入本文，你会欣然接受这个建议。

提一句，对于这类代码，函数式编程者往往喜欢使用更简短的 `=>` 箭头函数语法（请看第 2 章的 “语法” 小节），像这样：

```
var partial =
    (fn, ...presetArgs) =>
        (...laterArgs) =>
            fn( ...presetArgs, ...laterArgs );
```

毫无疑问这更加简洁，甚至代码稀少。但我个人觉得，无论我们从数学符号的对称性上获得什么好处，都会因函数变成了匿名函数而在整体的可读性上失去更多益处。此外，由于作用域边界变得模糊，我们会更加难以辩认闭包。

不管你喜欢哪种语法实现方式，现在我们用 `partial(..)` 实用函数来制造这些之前提及的偏函数：

```
var getPerson = partial( ajax, "http://some.api/person" );

var getOrder = partial( ajax, "http://some.api/order" );
```

请暂停并思考一下 `getPerson(..)` 函数的外形和内在。它相当于下面这样：

```
var getPerson = function partiallyApplied(...laterArgs) {
    return ajax( "http://some.api/person", ...laterArgs );
};
```

创建 `getOrder(..)` 函数可以依葫芦画瓢。但是 `getCurrentUser(..)` 函数又如何呢？

```
// 版本 1
var getCurrentUser = partial(
    ajax,
    "http://some.api/person",
    { user: CURRENT_USER_ID }
);

// 版本 2
var getCurrentUser = partial( getPerson, { user: CURRENT_USER_ID } );
```

我们可以（版本 1）直接通过指定 `url` 和 `data` 两个实参来定义 `getCurrentUser(..)` 函数，也可以（版本 2）将 `getCurrentUser(..)` 函数定义成 `getPerson(..)` 的偏应用，该偏应用仅指定一个附加的 `data` 实参。

因为版本 2 重用了已经定义好的函数，所以它在表达上更清晰一些。因此我认为它更加贴合函数式编程精神。

版本 1 和 2 分别相当于下面的代码，我们仅用这些代码来确认一下对两个函数版本内部运行机制的理解。

```
// 版本 1
var getCurrentUser = function partiallyApplied(...laterArgs) {
    return ajax(
        "http://some.api/person",
        { user: CURRENT_USER_ID },
        ...laterArgs
    );
};

// 版本 2
var getCurrentUser = function outerPartiallyApplied(...outerLaterArgs) {
    var getPerson = function innerPartiallyApplied(...innerLaterArgs){
        return ajax( "http://some.api/person", ...innerLaterArgs );
    };

    return getPerson( { user: CURRENT_USER_ID }, ...outerLaterArgs );
}
```

再强调一下，为了确保你理解这些代码段发生了什么，请暂停并重新阅读一下它们。

注意： 第二个版本的函数包含了一个额外的函数包装层。这看起来有些奇怪而且多余，但对于你真正要适应的函数式编程来说，这仅仅是它的冰山一角。随着本文的继续深入，我们将会把许多函数互相包装起来。记住，这就是函数式编程！

我们接着看另外一个偏应用的实用示例。设想一个 `add(..)` 函数，它接收两个实参，并取二者之和：

```
function add(x,y) {
    return x + y;
}
```

现在，想象我们要拿到一个数字列表，并且给其中每个数字加一个确定的数值。我们将使用 JS 数组对象内置的 `map(..)` 实用函数。

```
[1,2,3,4,5].map( function adder(val){
    return add( 3, val );
} );
// [4,5,6,7,8]
```

注意： 如果你没见过 `map(..)` ，别担心，我们会在本书后面的部分详细介绍它。目前你只需要知道它用来循环遍历（loop over）一个数组，在遍历过程中调用函数产出新值并存到新的数组中。

因为 `add(..)` 函数签名不是 `map(..)` 函数所预期的，所以我们不直接把它传入 `map(..)` 函数里。这样一来，偏应用就有了用武之地：我们可以调整 `add(..)` 函数签名，以符合 `map(..)` 函数的预期。

```
[1,2,3,4,5].map( partial( add, 3 ) );
// [4,5,6,7,8]
```

`bind(..)`

JavaScript 有一个内建的 `bind(..)` 实用函数，任何函数都可以使用它。该函数有两个功能：预设 `this` 关键字的上下文，以及偏应用实参。

我认为将这两个功能混合进一个实用函数是极其糟糕的决定。有时你不想关心 `this` 的绑定，而只是要偏应用实参。我本人基本上从不会同时需要这两个功能。

对于下面的方案，你通常要传 `null` 给用来绑定 `this` 的实参（第一个实参），而它是一个可以忽略的占位符。因此，这个方案非常糟糕。

请看:

```
var getPerson = ajax.bind( null, "http://some.api/person" );
```

那个 `null` 只会给我带来无尽的烦恼。

将实参顺序颠倒

回想我们之前调用 Ajax 函数的方式：`ajax( url, data, cb )`。如果要偏应用 `cb` 而稍后再指定 `data` 和 `url` 参数，我们应该怎么做呢？我们可以创建一个可以颠倒实参顺序的实用函数，用来包装原函数。

```
function reverseArgs(fn) {
    return function argsReversed(...args){
        return fn( ...args.reverse() );
    };
}

// ES6 箭头函数形式
var reverseArgs =
    fn =>
        (...args) =>
            fn( ...args.reverse() );
```

现在可以颠倒 `ajax(..)` 实参的顺序了，接下来，我们不再从左边开始，而是从右侧开始偏应用实参。为了恢复期望的实参顺序，接着我们又将偏应用实参后的函数颠倒一下实参顺序：

```
var cache = {};

var cacheResult = reverseArgs(
    partial( reverseArgs( ajax ), function onResult(obj){
        cache[obj.id] = obj;
    } )
);

// 处理后:
cacheResult( "http://some.api/person", { user: CURRENT_USER_ID } );
```

好，我们来定义一个从右边开始偏应用实参（译者注：以下简称右偏应用实参）的 `partialRight(..)` 实用函数。我们将运用和上面相同的技巧于该函数中：

```
function partialRight( fn, ...presetArgs ) {
    return reverseArgs(
        partial( reverseArgs( fn ), ...presetArgs.reverse() )
    );
}

var cacheResult = partialRight( ajax, function onResult(obj){
    cache[obj.id] = obj;
});

// 处理后:
cacheResult( "http://some.api/person", { user: CURRENT_USER_ID } );
```

这个 `partialRight(..)` 函数的实现方案不能保证让一个特定的形参接收特定的被偏应用的值；它只能确保将被这些值（一个或几个）当作原函数最右边的实参（一个或几个）传入。

举个例子:

```
function foo(x,y,z) {
    var rest = [].slice.call( arguments, 3 );
    console.log( x, y, z, rest );
}

var f = partialRight( foo, "z:last" );

f( 1, 2 );            // 1 2 "z:last" []

f( 1 );               // 1 "z:last" undefined []

f( 1, 2, 3 );         // 1 2 3 ["z:last"]

f( 1, 2, 3, 4 );      // 1 2 3 [4,"z:last"]
```

只有在传两个实参（匹配到 `x` 和 `y` 形参）调用 `f(..)` 函数时，`"z:last"` 这个值才能被赋给函数的形参 `z`。在其他的例子里，不管左边有多少个实参，`"z:last"` 都被传给最右的实参。

一次传一个

我们来看一个跟偏应用类似的技术，该技术将一个期望接收多个实参的函数拆解成连续的链式函数（chained functions），每个链式函数接收单一实参（实参个数：1）并返回另一个接收下一个实参的函数。

这就是柯里化（currying）技术。

首先，想象我们已创建了一个 `ajax(..)` 的柯里化版本。我们这样使用它：

```
curriedAjax( "http://some.api/person" )
    ( { user: CURRENT_USER_ID } )
        ( function foundUser(user){ /* .. */ } );
```

我们将三次调用分别拆解开来，这也许有助于我们理解整个过程：

```
var personFetcher = curriedAjax( "http://some.api/person" );

var getCurrentUser = personFetcher( { user: CURRENT_USER_ID } );

getCurrentUser( function foundUser(user){ /* .. */ } );
```

该 `curriedAjax(..)` 函数在每次调用中，一次只接收一个实参，而不是一次性接收所有实参（像 `ajax(..)` 那样），也不是先传部分实参再传剩余部分实参（借助 `partial(..)` 函数）。

柯里化和偏应用相似，每个类似偏应用的连续柯里化调用都把另一个实参应用到原函数，一直到所有实参传递完毕。

不同之处在于，`curriedAjax(..)` 函数会明确地返回一个期望只接收下一个实参 `data` 的函数（我们把它叫做 `curriedGetPerson(..)`），而不是那个能接收所有剩余实参的函数(像此前的 `getPerson(..)` 函数)。

如果一个原函数期望接收 5 个实参，这个函数的柯里化形式只会接收第一个实参，并且返回一个用来接收第二个参数的函数。而这个被返回的函数又只接收第二个参数，并且返回一个接收第三个参数的函数。依此类推。

由此而知，柯里化将一个多参数（higher-arity）函数拆解为一系列的单元链式函数。

如何定义一个用来柯里化的实用函数呢？我们将要用到第 2 章中的一些技巧。

```
function curry(fn,arity = fn.length) {
    return (function nextCurried(prevArgs){
        return function curried(nextArg){
            var args = prevArgs.concat( [nextArg] );

            if (args.length >= arity) {
                return fn( ...args );
            }
            else {
                return nextCurried( args );
            }
        };
    })( [] );
}
```

ES6 箭头函数版本：

```
var curry =
    (fn, arity = fn.length, nextCurried) =>
        (nextCurried = prevArgs =>
            nextArg => {
                var args = prevArgs.concat( [nextArg] );

                if (args.length >= arity) {
                    return fn( ...args );
                }
                else {
                    return nextCurried( args );
                }
            }
        )( [] );
```

此处的实现方式是把空数组 `[]` 当作 `prevArgs` 的初始实参集合，并且将每次接收到的 `nextArg` 同 `prevArgs` 连接成 `args` 数组。当 `args.length` 小于 `arity`（原函数 `fn(..)` 被定义和期望的形参数量）时，返回另一个 `curried(..)` 函数（译者注：这里指代 `nextCurried(..)` 返回的函数）用来接收下一个 `nextArg` 实参，与此同时将 `args` 实参集合作为唯一的 `prevArgs` 参数传入 `nextCurried(..)` 函数。一旦我们收集了足够长度的 `args` 数组，就用这些实参触发原函数 `fn(..)` 。

默认地，我们的实现方案基于下面的条件：在拿到原函数期望的全部实参之前，我们能够通过检查将要被柯里化的函数的 `length` 属性来得知柯里化需要迭代多少次。

假如你将该版本的 `curry(..)` 函数用在一个 `length` 属性不明确的函数上 —— 函数的形参声明包含默认形参值、形参解构，或者它是可变参数函数，用 `...args` 当形参；参考第 2 章 —— 你将要传入 `arity` 参数（作为 `curry(..)` 的第二个形参）来确保 `curry(..)` 函数的正常运行。

我们用 `curry(..)` 函数来实现此前的 `ajax(..)` 例子：

```
var curriedAjax = curry( ajax );

var personFetcher = curriedAjax( "http://some.api/person" );

var getCurrentUser = personFetcher( { user: CURRENT_USER_ID } );

getCurrentUser( function foundUser(user){ /* .. */ } );
```

如上，我们每次函数调用都会新增一个实参，最终给原函数 `ajax(..)` 使用，直到收齐三个实参并执行 `ajax(..)` 函数为止。

还记得前面讲到为数值列表的每个值加 `3` 的那个例子吗？回顾一下，由于柯里化是和偏应用相似的，所以我们可以用几乎相同的方式以柯里化来完成那个例子。

```
[1,2,3,4,5].map( curry( add )( 3 ) );
// [4,5,6,7,8]
```

`partial(add,3)` 和 `curry(add)(3)` 两者有什么不同呢？为什么你会选 `curry(..)` 而不是偏函数呢？当你先得知 `add(..)` 是将要被调整的函数，但如果这个时候并不能确定 `3` 这个值，柯里化可能会起作用：

```
var adder = curry( add );

// later
[1,2,3,4,5].map( adder( 3 ) );
// [4,5,6,7,8]
```

让我们来看看另一个有关数字的例子，这次我们拿一个列表的数字做加法：

```
function sum(...args) {
    var sum = 0;
    for (let i = 0; i < args.length; i++) {
        sum += args[i];
    }
    return sum;
}

sum( 1, 2, 3, 4, 5 );                       // 15

// 好，我们看看用柯里化怎么做：
// (5 用来指定需要链式调用的次数)
var curriedSum = curry( sum, 5 );

curriedSum( 1 )( 2 )( 3 )( 4 )( 5 );        // 15
```

这里柯里化的好处是，每次函数调用传入一个实参，并生成另一个特定性更强的函数，之后我们可以在程序中获取并使用那个新函数。而偏应用则是预先指定所有将被偏应用的实参，产出一个等待接收剩下所有实参的函数。

如果想用偏应用来每次指定一个形参，你得在每个函数中逐次调用 `partialApply(..)` 函数。而被柯里化的函数可以自动完成这个工作，这让一次单独传递一个参数变得更加符合人机工程学。

在 JavaScript 中，柯里化和偏应用都使用闭包来保存实参，直到收齐所有实参后我们再执行原函数。

柯里化和偏应用有什么用？

无论是柯里化风格（`sum(1)(2)(3)`）还是偏应用风格（`partial(sum,1,2)(3)`），它们的签名比普通函数签名奇怪得多。那么，在适应函数式编程的时候，我们为什么要这么做呢？答案有几个方面。

首先是显而易见的理由，使用柯里化和偏应用可以将指定分离实参的时机和地方独立开来（遍及代码的每一处），而传统函数调用则需要预先确定所有实参。如果你在代码某一处只获取了部分实参，然后在另一处确定另一部分实参，这个时候柯里化和偏应用就能派上用场。

另一个最能体现柯里化应用的的是，当函数只有一个形参时，我们能够比较容易地组合它们。因此，如果一个函数最终需要三个实参，那么它被柯里化以后会变成需要三次调用，每次调用需要一个实参的函数。当我们组合函数时，这种单元函数的形式会让我们处理起来更简单。我们将在后面继续探讨这个话题。

如何柯里化多个实参？

到目前为止，我相信我给出的是我们能在 JavaScript 中能得到的，最精髓的柯里化定义和实现方式。

具体来说，如果简单看下柯里化在 Haskell 语言中的应用，我们会发现一个函数总是在一次柯里化调用中接收多个实参 —— 而不是接收一个包含多个值的元组（tuple，类似我们的数组）实参。

在 Haskell 中的示例：

```
foo 1 2 3
```

该示例调用了 `foo` 函数，并且根据传入的三个值 `1`、`2` 和 `3` 得到了结果。但是在 Haskell 中，函数会自动被柯里化，这意味着我们传入函数的值都分别传入了单独的柯里化调用。在 JS 中看起来则会是这样：`foo(1)(2)(3)`。这和我此前讲过的 `curry(..)` 风格如出一辙。

注意： 在 Haskell 中，`foo (1,2,3)` 不是把三个值当作单独的实参一次性传入函数，而是把它们包含在一个元组（类似 JS 数组）中作为单独实参传入函数。为了正常运行，我们需要改变 `foo` 函数来处理作为实参的元组。据我所知，在 Haskell 中我们没有办法在一次函数调用中将全部三个实参独立地传入，而需要柯里化调用每个函数。诚然，多次调用对于 Haskell 开发者来说是透明的，但对 JS 开发者来说，这在语法上更加一目了然。

基于以上原因，我认为此前展示的 `curry(..)` 函数是一个对 Haskell 柯里化的可靠改编，我把它叫做 “严格柯里化”。

然而，我们需要注意，大多数流行的 JavaScript 函数式编程库都使用了一种并不严格的柯里化（loose currying）定义。

具体来说，往往 JS 柯里化实用函数会允许你在每次柯里化调用中指定多个实参。回顾一下之前提到的 `sum(..)` 示例，松散柯里化应用会是下面这样：

```
var curriedSum = looseCurry( sum, 5 );

curriedSum( 1 )( 2, 3 )( 4, 5 );            // 15
```

可以看到，语法上我们节省了 `()` 的使用，并且把五次函数调用减少成三次，间接提高了性能。除此之外，使用 `looseCurry(..)` 函数的结果也和之前更加狭义的 `curry(..)` 函数一样。我猜便利性和性能因素是众框架允许多实参柯里化的原因。这看起来更像是品味问题。

注意： 松散柯里化允许你传入超过形参数量（arity，原函数确认或指定的形参数量）的实参。如果你将函数的参数设计成可配的或变化的，那么松散柯里化将会有利于你。例如，如果你将要柯里化的函数接收 5 个实参，松散柯里化依然允许传入超过 5 个的实参（`curriedSum(1)(2,3,4)(5,6)`），而严格柯里化就不支持 `curriedSum(1)(2)(3)(4)(5)(6)`。

我们可以将之前的柯里化实现方式调整一下，使其适应这种常见的更松散的定义：

```
function looseCurry(fn,arity = fn.length) {
    return (function nextCurried(prevArgs){
        return function curried(...nextArgs){
            var args = prevArgs.concat( nextArgs );

            if (args.length >= arity) {
                return fn( ...args );
            }
            else {
                return nextCurried( args );
            }
        };
    })( [] );
}
```

现在每个柯里化调用可以接收一个或多个实参了（收集在 `nextArgs` 数组中）。至于这个实用函数的 ES6 箭头函数版本，我们就留作一个小练习，有兴趣的读者可以模仿之前 `curry(..)` 函数的来完成。

反柯里化

你也会遇到这种情况：拿到一个柯里化后的函数，却想要它柯里化之前的版本 —— 这本质上就是想将类似 `f(1)(2)(3)` 的函数变回类似 `g(1,2,3)` 的函数。

不出意料的话，处理这个需求的标准实用函数通常被叫作 `uncurry(..)`。下面是简陋的实现方式：

```
function uncurry(fn) {
    return function uncurried(...args){
        var ret = fn;

        for (let i = 0; i < args.length; i++) {
            ret = ret( args[i] );
        }

        return ret;
    };
}

// ES6 箭头函数形式
var uncurry =
    fn =>
        (...args) => {
            var ret = fn;

            for (let i = 0; i < args.length; i++) {
                ret = ret( args[i] );
            }

            return ret;
        };
```

警告：请不要以为 `uncurry(curry(f))` 和 `f` 函数的行为完全一样。虽然在某些库中，反柯里化使函数变成和原函数（译者注：这里的原函数指柯里化之前的函数）类似的函数，但是凡事皆有例外，我们这里就有一个例外。如果你传入原函数期望数量的实参，那么在反柯里化后，函数的行为（大多数情况下）和原函数相同。然而，如果你少传了实参，就会得到一个仍然在等待传入更多实参的部分柯里化函数。我们在下面的代码中说明这个怪异行为。

```
function sum(...args) {
    var sum = 0;
    for (let i = 0; i < args.length; i++) {
        sum += args[i];
    }
    return sum;
}

var curriedSum = curry( sum, 5 );
var uncurriedSum = uncurry( curriedSum );

curriedSum( 1 )( 2 )( 3 )( 4 )( 5 );        // 15

uncurriedSum( 1, 2, 3, 4, 5 );              // 15
uncurriedSum( 1, 2, 3 )( 4 )( 5 );          // 15
```

`uncurry()` 函数最为常见的作用对象很可能并不是人为生成的柯里化函数（例如上文所示），而是某些操作所产生的已经被柯里化了的结果函数。我们将在本章后面关于“无形参风格”的讨论中阐述这种应用场景。

只要一个实参

设想你向一个实用函数传入一个函数，而这个实用函数会把多个实参传入函数，但可能你只希望你的函数接收单一实参。如果你有个类似我们前面提到被松散柯里化的函数，它能接收多个实参，但你却想让它接收单一实参。那么这就是我想说的情况。

我们可以设计一个简单的实用函数，它包装一个函数调用，确保被包装的函数只接收一个实参。既然实际上我们是强制把一个函数处理成单参数函数（unary），那我们索性就这样命名实用函数：

```
function unary(fn) {
    return function onlyOneArg(arg){
        return fn( arg );
    };
}

// ES6 箭头函数形式
var unary =
    fn =>
        arg =>
            fn( arg );
```

我们此前已经和 `map(..)` 函数打过照面了。它调用传入其中的 mapping 函数时会传入三个实参：`value`、`index` 和 `list`。如果你希望你传入 `map(..)` 的 mapping 函数只接收一个参数，比如 `value`，你可以使用 `unary(..)` 函数来操作：

```
function unary(fn) {
    return function onlyOneArg(arg){
        return fn( arg );
    };
}

var adder = looseCurry( sum, 2 );

// 出问题了：
[1,2,3,4,5].map( adder( 3 ) );
// ["41,2,3,4,5", "61,2,3,4,5", "81,2,3,4,5", "101, ...

// 用 `unary(..)` 修复后:
[1,2,3,4,5].map( unary( adder( 3 ) ) );
// [4,5,6,7,8]
```

另一种常用的 `unary(..)` 函数调用示例：

```
["1","2","3"].map( parseFloat );
// [1,2,3]

["1","2","3"].map( parseInt );
// [1,NaN,NaN]

["1","2","3"].map( unary( parseInt ) );
// [1,2,3]
```

对于 `parseInt(str,radix)` 这个函数调用，如果 `map(..)` 函数调用它时在它的第二个实参位置传入 `index`，那么毫无疑问 `parseInt(..)` 会将 `index` 理解为 `radix` 参数，这是我们不希望发生的。而 `unary(..)` 函数创建了一个只接收第一个传入实参，忽略其他实参的新函数，这就意味着传入 `index` 不再会被误解为 `radix` 参数。

传一个返回一个

说到只传一个实参的函数，在函数式编程工具库中有另一种通用的基础函数：该函数接收一个实参，然后什么都不做，原封不动地返回实参值。

```
function identity(v) {
    return v;
}

// ES6 箭头函数形式
var identity =
    v =>
        v;
```

看起来这个实用函数简单到了无处可用的地步。但即使是简单的函数在函数式编程的世界里也能发挥作用。就像演艺圈有句谚语：没有小角色，只有小演员。

举个例子，想象一下你要用正则表达式拆分（split up）一个字符串，但输出的数组中可能包含一些空值。我们可以使用 `filter(..)` 数组方法（下文会详细说到这个方法）来筛除空值，而我们将 `identity(..)` 函数作为 `filter(..)` 的断言：

```
var words = "   Now is the time for all...  ".split( /\s|\b/ );
words;
// ["","Now","is","the","time","for","all","...",""]

words.filter( identity );
// ["Now","is","the","time","for","all","..."]
```

既然 `identity(..)` 会简单地返回传入的值，而 JS 会将每个值强制转换为 `true` 或 `false`，这样我们就能在最终的数组里对每个值进行保存或排除。

小贴士：像这个例子一样，另外一个能被用作断言的单实参函数是 JS 自有的 `Boolean(..)` 方法，该方法会强制把传入值转为 `true` 或 `false`。

另一个使用 `identity(..)` 的示例就是将其作为替代一个转换函数（译者注：transformation，这里指的是对传入值进行修改或调整，返回新值的函数）的默认函数：

```
function output(msg,formatFn = identity) {
    msg = formatFn( msg );
    console.log( msg );
}

function upper(txt) {
    return txt.toUpperCase();
}

output( "Hello World", upper );        // HELLO WORLD
output( "Hello World" );            // Hello World
```

如果不给 `output(..)` 函数的 `formatFn` 参数设置默认值，我们可以叫出老朋友 `partialRight(..)` 函数：

```
var specialOutput = partialRight( output, upper );
var simpleOutput = partialRight( output, identity );

specialOutput( "Hello World" );        // HELLO WORLD
simpleOutput( "Hello World" );        // Hello World
```

你也可能会看到 `identity(..)` 被当作 `map(..)` 函数调用的默认转换函数，或者作为某个函数数组的 `reduce(..)` 函数的初始值。我们将会在第 8 章中提到这两个实用函数。

恒定参数

Certain API 禁止直接给方法传值，而要求我们传入一个函数，就算这个函数只是返回一个值。JS Promise 中的 `then(..)` 方法就是一个 Certain API。很多人声称 ES6 箭头函数可以当作这个问题的“解决方案”。但我这有一个函数式编程实用函数可以完美胜任该任务：

```
function constant(v) {
    return function value(){
        return v;
    };
}

// or the ES6 => form
var constant =
    v =>
        () =>
            v;
```

这个微小而简洁的实用函数可以解决我们关于 `then(..)` 的烦恼：

```
p1.then( foo ).then( () => p2 ).then( bar );

// 对比：

p1.then( foo ).then( constant( p2 ) ).then( bar );
```

警告： 尽管使用 `() => p2` 箭头函数的版本比使用 `constant(p2)` 的版本更简短，但我建议你忍住别用前者。该箭头函数返回了一个来自外作用域的值，这和 函数式编程的理念有些矛盾。我们将会在后面第 5 章的“减少副作用”小节中提到这种行为带来的陷阱。

扩展在参数中的妙用

在第 2 章中，我们简要地讲到了形参数组解构。回顾一下该示例：

```
function foo( [x,y,...args] ) {
    // ..
}

foo( [1,2,3] );
```

在 `foo(..)` 函数的形参列表中，我们期望接收单一数组实参，我们要把这个数组拆解 —— 或者更贴切地说，扩展（spread out）—— 成独立的实参 `x` 和 `y` 。除了头两个位置以外的参数值我们都会通过 `...` 操作将它们收集在 `args` 数组中。

当函数必须接收一个数组，而你却想把数组内容当成单独形参来处理的时候，这个技巧十分有用。

然而，有的时候，你无法改变原函数的定义，但想使用形参数组解构。举个例子，请思考下面的函数：

```
function foo(x,y) {
    console.log( x + y );
}

function bar(fn) {
    fn( [ 3, 9 ] );
}

bar( foo );            // 失败
```

你注意到为什么 `bar(foo)` 函数失败了吗？

我们将 `[3,9]` 数组作为单一值传入 `fn(..)` 函数，但 `foo(..)` 期望接收单独的 `x` 和 `y` 形参。如果我们可以把 `foo(..)` 的函数声明改变成 `function foo([x,y]) { ..` 那就好办了。或者，我们可以改变 `bar(..)` 函数的行为，把调用改成 `fn(...[3,9])`，这样就能将 `3` 和 `9` 分别传入 `foo(..)` 函数了。

假设有两个在此方法上互不兼容的函数，而且由于各种原因你无法改变它们的声明和定义。那么你该如何一并使用它们呢？

为了调整一个函数，让它能把接收的单一数组扩展成各自独立的实参，我们可以定义一个辅助函数：

```
function spreadArgs(fn) {
    return function spreadFn(argsArr) {
        return fn( ...argsArr );
    };
}

// ES6 箭头函数的形式：
var spreadArgs =
    fn =>
        argsArr =>
            fn( ...argsArr );
```

注意： 我把这个辅助函数叫做 `spreadArgs(..)`，但一些库，比如 Ramda，经常把它叫做 `apply(..)`。

现在我们可以使用 `spreadArgs(..)` 来调整 `foo(..)` 函数，使其作为一个合适的输入参数并正常地工作：

```
bar( spreadArgs( foo ) );            // 12
```

相信我，虽然我不能讲清楚这些问题出现的原因，但它们一定会出现的。本质上，`spreadArgs(..)` 函数使我们能够定义一个借助数组 `return` 多个值的函数，不过，它让这些值仍然能分别作为其他函数的输入参数来处理。

一个函数的输出作为另外一个函数的输入被称作组合（composition），我们将在第四章详细讨论这个话题。

尽管我们在谈论 `spreadArgs(..)` 实用函数，但我们也可以定义一下实现相反功能的实用函数：

```
function gatherArgs(fn) {
    return function gatheredFn(...argsArr) {
        return fn( argsArr );
    };
}

// ES6 箭头函数形式
var gatherArgs =
    fn =>
        (...argsArr) =>
            fn( argsArr );
```

注意： 在 Ramda 中，该实用函数被称作 `unapply(..)`，是与 `apply(..)` 功能相反的函数。我认为术语 "扩展（spread）" 和 "聚集（gather）" 可以把这两个函数发生的事情解释得更好一些。

因为有时我们可能要调整一个函数，解构其数组形参，使其成为另一个分别接收单独实参的函数，所以我们可以通过使用 `gatherArgs(..)` 实用函数来将单独的实参聚集到一个数组中。我们将在第 8 章中细说 `reduce(..)` 函数，这里我们简要说一下：它重复调用传入的 reducer 函数，其中 reducer 函数有两个形参，现在我们可以将这两个形参聚集起来：

```
function combineFirstTwo([ v1, v2 ]) {
    return v1 + v2;
}

[1,2,3,4,5].reduce( gatherArgs( combineFirstTwo ) );
// 15
```

参数顺序的那些事儿

对于多形参函数的柯里化和偏应用，我们不得不通过许多令人懊恼的技巧来修正这些形参的顺序。有时我们把一个函数的形参顺序定义成柯里化需求的形参顺序，但这种顺序没有兼容性，我们不得不绞尽脑汁来重新调整它。

让人沮丧的可不仅是我们需要使用实用函数来委曲求全，在此之外，这种做法还会导致我们的代码被无关代码混淆。这种东西就像碎纸片，这一片那一片的，而不是一整个突出问题，但这些问题的细碎丝毫不会减少它们带来的苦恼。

难道就没有能让我们从修正参数顺序这件事里解脱出来的方法吗！？

在第 2 章里，我们讲到了命名实参（named-argument）解构模式。回顾一下：

```
function foo( {x,y} = {} ) {
    console.log( x, y );
}

foo( {
    y: 3
} );                    // undefined 3
```

我们将 `foo(..)` 函数的第一个形参 —— 它被期望是一个对象 —— 解构成单独的形参 `x` 和 `y`。接着在调用时传入一个对象实参，并且提供函数期望的属性，这样就可以把“命名实参”映射到相应形参上。

命名实参主要的好处就是不用再纠结实参传入的顺序，因此提高了可读性。我们可以发掘一下看看是否能设计一个等效的实用函数来处理对象属性，以此提高柯里化和偏应用的可读性：

```
function partialProps(fn,presetArgsObj) {
    return function partiallyApplied(laterArgsObj){
        return fn( Object.assign( {}, presetArgsObj, laterArgsObj ) );
    };
}

function curryProps(fn,arity = 1) {
    return (function nextCurried(prevArgsObj){
        return function curried(nextArgObj = {}){
            var [key] = Object.keys( nextArgObj );
            var allArgsObj = Object.assign( {}, prevArgsObj, { [key]: nextArgObj[key]
} );

            if (Object.keys( allArgsObj ).length >= arity) {
                return fn( allArgsObj );
            }
            else {
                return nextCurried( allArgsObj );
            }
        };
    })( {} );
}
```

我们甚至不需要设计一个 `partialPropsRight(..)` 函数了，因为我们根本不需要考虑属性的映射顺序，通过命名来映射形参完全解决了我们有关于顺序的烦恼！

我们这样使用这些使用函数：

```
function foo({ x, y, z } = {}) {
    console.log( `x:${x} y:${y} z:${z}` );
}

var f1 = curryProps( foo, 3 );
var f2 = partialProps( foo, { y: 2 } );

f1( {y: 2} )( {x: 1} )( {z: 3} );
// x:1 y:2 z:3

f2( { z: 3, x: 1 } );
// x:1 y:2 z:3
```

我们不用再为参数顺序而烦恼了！现在，我们可以指定我们想传入的实参，而不用管它们的顺序如何。再也不需要类似 `reverseArgs(..)` 的函数或其它妥协了。赞！

属性扩展

不幸的是，只有在我们可以掌控 `foo(..)` 的函数签名，并且可以定义该函数的行为，使其解构第一个参数的时候，以上技术才能起作用。如果一个函数，其形参是各自独立的（没有经过形参解构），而且不能改变它的函数签名，那我们应该如何运用这个技术呢？

```
function bar(x,y,z) {
    console.log( `x:${x} y:${y} z:${z}` );
}
```

就像之前的 `spreadArgs(..)` 实用函数一样，我们也可以定义一个 `spreadArgProps(..)` 辅助函数，它接收对象实参的 `key: value` 键值对，并将其“扩展”成独立实参。

不过，我们需要注意某些异常的地方。我们使用 `spreadArgs(..)` 函数处理数组实参时，参数的顺序是明确的。然而，对象属性的顺序是不太明确且不可靠的。取决于不同对象的创建方式和属性设置方式，我们无法完全确认对象会产生什么顺序的属性枚举。

针对这个问题，我们定义的实用函数需要让你能够指定函数期望的实参顺序（比如属性枚举的顺序）。我们可以传入一个类似 `["x","y","z"]` 的数组，通知实用函数基于该数组的顺序来获取对象实参的属性值。

这着实不错，但还是有点瑕疵，就算是最简单的函数，我们也免不了为其增添一个由属性名构成的数组。难道我们就没有一种可以探知函数形参顺序的技巧吗？哪怕给一个普通而简单的例子？还真有！

JavaScript 的函数对象上有一个 `.toString()` 方法，它返回函数代码的字符串形式，其中包括函数声明的签名。先忽略其正则表达式分析技巧，我们可以通过解析函数字符串来获取每个单独的命名形参。虽然这段代码看起来有些粗暴，但它足以满足我们的需求：

```
function spreadArgProps(
    fn,
    propOrder =
        fn.toString()
        .replace( /^(?:(?:function.*\(([^]*?)\))|(?:([^\(\)]+?)\s*=>)|(?:\(([^]*?)\)\s
*=>))[^]+$/, "$1$2$3" )
        .split( /\s*,\s*/ )
        .map( v => v.replace( /[=\s].*$/, "" ) )
) {
    return function spreadFn(argsObj) {
        return fn( ...propOrder.map( k => argsObj[k] ) );
    };
}
```

注意： 该实用函数的参数解析逻辑并非无懈可击，使用正则来解析代码这个前提就已经很不靠谱了！但处理一般情况是我们的唯一目标，从这点来看这个实用函数还是恰到好处的。我们需要的只是对简单形参（包括带默认值的形参）函数的形参顺序做一个恰当的默认检测。例如，我们的实用函数不需要把复杂的解构形参给解析出来，因为无论如何我们不太可能对拥有这种复杂形参的函数使用 `spreadArgProps()` 函数。因此该逻辑能搞定 80% 的需求，它允许我们在其它不能正确解析复杂函数签名的情况下覆盖 `propOrder` 数组形参。这是本书尽可能寻找的一种实用性平衡。

让我们看看 `spreadArgProps(..)` 实用函数是怎么用的：

```
function bar(x,y,z) {
    console.log( `x:${x} y:${y} z:${z}` );
}

var f3 = curryProps( spreadArgProps( bar ), 3 );
var f4 = partialProps( spreadArgProps( bar ), { y: 2 } );

f3( {y: 2} )( {x: 1} )( {z: 3} );
// x:1 y:2 z:3

f4( { z: 3, x: 1 } );
// x:1 y:2 z:3
```

提个醒：本文中呈现的对象形参（object parameters）和命名实参（named arguments）模式，通过减少由调整实参顺序带来的干扰，明显地提高了代码的可读性，不过据我所知，没有哪个主流的函数式编程库使用该方案。所以你会看到该做法与大多数 JavaScript 函数式编程很不一样.

此外，使用在这种风格下定义的函数要求你知道每个实参的名字。你必须记住：“这个函数形参叫作 ‘fn’ ”，而不是只记得：“噢，把这个函数作为第一个实参传进去”。

请小心地权衡它们。

无形参风格

在函数式编程的世界中，有一种流行的代码风格，其目的是通过移除不必要的形参-实参映射来减少视觉上的干扰。这种风格的正式名称为 “隐性编程（tacit programming）”，一般则称作 “无形参（point-free）” 风格。术语 “point” 在这里指的是函数形参。

警告： 且慢，先说明我们这次的讨论是一个有边界的提议，我不建议你在函数式编程的代码里不惜代价地滥用无形参风格。该技术是用于在适当情况下提升可读性。但你完全可能像滥用软件开发里大多数东西一样滥用它。如果你由于必须迁移到无参数风格而让代码难以理解，请打住。你不会因此获得小红花，因为你用看似聪明但晦涩难懂的方式抹除形参这个点的同时，还抹除了代码的重点。

我们从一个简单的例子开始：

```
function double(x) {
    return x * 2;
}

[1,2,3,4,5].map( function mapper(v){
    return double( v );
} );
// [2,4,6,8,10]
```

可以看到 `mapper(..)` 函数和 `double(..)` 函数有相同（或相互兼容）的函数签名。形参（也就是所谓的 “point“） `v` 可以直接映射到 `double(..)` 函数调用里相应的实参上。这样， `mapper(..)` 函数包装层是非必需的。我们可以将其简化为无形参风格：

```
function double(x) {
    return x * 2;
}

[1,2,3,4,5].map( double );
// [2,4,6,8,10]
```

回顾之前的一个例子：

```
["1","2","3"].map( function mapper(v){
    return parseInt( v );
} );
// [1,2,3]
```

该例中， `mapper(..)` 实际上起着重要作用，它排除了 `map(..)` 函数传入的 `index` 实参，因为如果不这么做的话， `parseInt(..)` 函数会错把 `index` 当作 `radix` 来进行整数解析。该例子中我们可以借助 `unary(..)` 函数：

```
["1","2","3"].map( unary( parseInt ) );
// [1,2,3]
```

使用无形参风格的关键，是找到你代码中，有哪些地方的函数直接将其形参作为内部函数调用的实参。以上提到的两个例子中， `mapper(..)` 函数拿到形参 `v` 单独传入了另一个函数调用。我们可以借助 `unary(..)` 函数将提取形参的逻辑层替换成无参数形式表达式。

警告： 你可能跟我一样，已经尝试着使用 `map(partialRight(parseInt,10))` 来将 `10` 右偏应用为 `parseInt(..)` 的 `radix` 实参。然而，就像我们之前看到的那样， `partialRight(..)` 仅仅保证将 `10` 当作最后一个实参传入原函数，而不是将其指定为第二个实参。因为 `map(..)` 函数本身会将 3 个实参（ `value` 、 `index` 和 `arr` ）传入它的映射函数，所以 `10` 就会被当成第四个实参传入 `parseInt(..)` 函数，而这个函数只会对头两个实参作出反应。

来看另一个例子：

```
// 将 `console.log` 当成一个函数使用
// 便于避免潜在的绑定问题

function output(txt) {
    console.log( txt );
}

function printIf( predicate, msg ) {
    if (predicate( msg )) {
        output( msg );
    }
}

function isShortEnough(str) {
    return str.length <= 5;
}

var msg1 = "Hello";
var msg2 = msg1 + " World";

printIf( isShortEnough, msg1 );            // Hello
printIf( isShortEnough, msg2 );
```

现在，我们要求当信息足够长时，将它打印出来，换而言之，我们需要一个 `!isShortEnough(..)` 断言。你可能会首先想到：

```
function isLongEnough(str) {
    return !isShortEnough( str );
}

printIf( isLongEnough, msg1 );
printIf( isLongEnough, msg2 );            // Hello World
```

这太简单了...但现在我们的重点来了！你看到了 `str` 形参是如何传递的吗？我们能否不通过重新实现 `str.length` 的检查逻辑，而重构代码并使其变成无形参风格呢？

我们定义一个 `not(..)` 取反辅助函数（在函数式编程库中又被称作 `complement(..)`）：

```
function not(predicate) {
    return function negated(...args){
        return !predicate( ...args );
    };
}

// ES6 箭头函数形式
var not =
    predicate =>
        (...args) =>
            !predicate( ...args );
```

接着，我们使用 `not(..)` 函数来定义无形参的 `isLongEnough(..)` 函数：

```
var isLongEnough = not( isShortEnough );

printIf( isLongEnough, msg2 );            // Hello World
```

目前为止已经不错了，但还能更进一步。我们实际上可以将 `printIf(..)` 函数本身重构成无形参风格。

我们可以用 `when(..)` 实用函数来表示 `if` 条件句：

```
function when(predicate,fn) {
    return function conditional(...args){
        if (predicate( ...args )) {
            return fn( ...args );
        }
    };
}

// ES6 箭头函数形式
var when =
    (predicate,fn) =>
        (...args) =>
            predicate( ...args ) ? fn( ...args ) : undefined;
```

我们把本章前面讲到的另一些辅助函数和 `when(..)` 函数结合起来搞定无形参风格的 `printIf(..)` 函数：

```
var printIf = uncurry( rightPartial( when, output ) );
```

我们是这么做的：将 `output` 方法右偏应用为 `when(..)` 函数的第二个（ `fn` 形参）实参，这样我们得到了一个仍然期望接收第一个实参（ `predicate` 形参）的函数。当该函数被调用时，会产生另一个期望接收（译者注：需要被打印的）信息字符串的函数，看起来就是这样： `fn(predicate)(str)` 。

多个（两个）链式函数的调用看起来很挫，就像被柯里化的函数。于是我们用 `uncurry(..)` 函数处理它，得到一个期望接收 `str` 和 `predicate` 两个实参的函数，这样该函数的签名就和 `printIf(predicate,str)` 原函数一样了。

我们把整个例子复盘一下（假设我们本章已经讲解的实用函数都在这里了）：

```
function output(msg) {
    console.log( msg );
}

function isShortEnough(str) {
    return str.length <= 5;
}

var isLongEnough = not( isShortEnough );

var printIf = uncurry( partialRight( when, output ) );

var msg1 = "Hello";
var msg2 = msg1 + " World";

printIf( isShortEnough, msg1 );            // Hello
printIf( isShortEnough, msg2 );

printIf( isLongEnough, msg1 );
printIf( isLongEnough, msg2 );            // Hello World
```

但愿无形参风格编程的函数式编程实践逐渐变得更有意义。你仍然可以通过大量实践来训练自己，让自己接受这种风格。再次提醒，请三思而后行，掂量一下是否值得使用无形参风格编程，以及使用到什么程度会益于提高代码的可读性。

有形参还是无形参，你怎么选？

注意： 还有什么无形参风格编程的实践呢？我们将在第 4 章的“回顾形参”小节里，站在新学习的组合函数知识之上来回顾这个技术。

总结

偏应用是用来减少函数的参数数量 —— 一个函数期望接收的实参数量 —— 的技术，它减少参数数量的方式是创建一个预设了部分实参的新函数。

柯里化是偏应用的一种特殊形式，其参数数量降低为 1，这种形式包含一串连续的链式函数调用，每个调用接收一个实参。当这些链式调用指定了所有实参时，原函数就会拿到收集好的实参并执行。你同样可以将柯里化还原。

其它类似 `unary(..)` 、 `identity(..)` 以及 `constant(..)` 的重要函数操作，是函数式编程基础工具库的一部分。

无形参是一种书写代码的风格，这种风格移除了非必需的形参映射实参逻辑，其目的在于提高代码的可读性和可理解性。

JavaScript轻量级函数式编程

第 4 章：组合函数

到目前为止，我希望你能更轻松地理解在函数式编程中使用函数意味着什么。

一个函数式编程者，会将他们程序中的每一个函数当成一小块简单的乐高部件。他们能一眼辨别出蓝色的 2x2 方块，并准确地知道它是如何工作的、能用它做些什么。当构建一个更大、更复杂的乐高模型时，当每一次需要下一块部件的时候，他们能够准确地从备用部件中找到这些部件并拿过来使用。

但有些时候，你把蓝色 2x2 的方块和灰色 4x1 的方块以某种形式组装到一起，然后意识到："这是个有用的部件，我可能会常用到它"。

那么你现在想到了一种新的"部件"，它是两种其他部件的组合，在需要的时候能触手可及。这时候，将这个蓝黑色 L 形状的方块组合体放到需要使用的地方，比每次分开考虑两种独立方块的组合要有效的多。

函数有多种多样的形状和大小。我们能够定义某种组合方式，来让它们成为一种新的组合函数，程序中不同的部分都可以使用这个函数。这种将函数一起使用的过程叫做组合。

输出到输入

我们已经见过几种组合的例子。比如，在第 3 章中，我们对 `unary(..)` 的讨论包含了如下表达式：`unary(adder(3))`。仔细想想这里发生了什么。

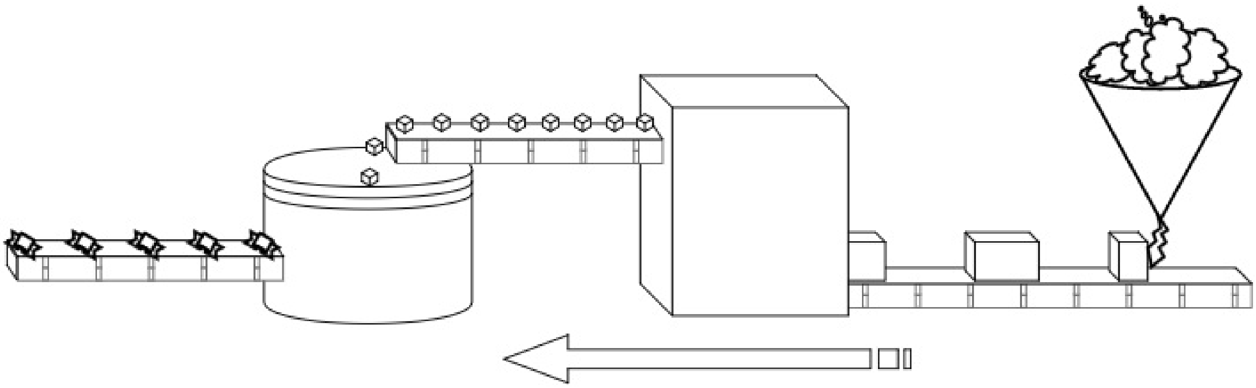
为了将两个函数整合起来，将第一个函数调用产生的输出当做第二个函数调用的输入。在 `unary(adder(3))` 中，`adder(3)` 的调用返回了一个值（值是一个函数）；该值被直接作为一个参数传入到 `unary(..)` 中，同样的，这个调用返回了一个值（值为另一个函数）。

让我们回放一下过程并且将数据流动的概念视觉化，是这个样子：

```
functionValue <-- unary <-- adder <-- 3
```

`3` 是 `adder(..)` 的输入。而 `adder(..)` 的输出是 `unary(..)` 的输入。`unary(..)` 的输出是 `functionValue`。 这就是 `unary(..)` 和 `adder(..)` 的组合。

把数据的流向想象成糖果工厂的一条传送带，每一次操作其实都是冷却、切割、包装糖果中的一步。在该章节中，我们将会用糖果工厂的类比来解释什么是组合。



让我们一步一步的来了解组合。首先假设你程序中可能存在这么两个实用函数。

```
function words(str) {
    return String( str )
        .toLowerCase()
        .split( /\s|\b/ )
        .filter( function alpha(v){
            return /^[\w]+$/.test( v );
        } );
}

function unique(list) {
    var uniqList = [];

    for (let i = 0; i < list.length; i++) {
        // value not yet in the new list?
        if (uniqList.indexOf( list[i] ) === -1 ) {
            uniqList.push( list[i] );
        }
    }

    return uniqList;
}
```

使用这两个实用函数来分析文本字符串：

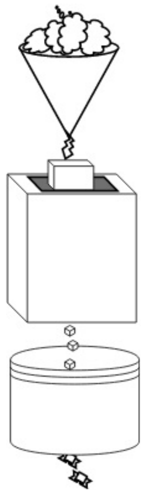
```
var text = "To compose two functions together, pass the \
output of the first function call as the input of the \
second function call.";

var wordsFound = words( text );
var wordsUsed = unique( wordsFound );

wordsUsed;
// ["to","compose","two","functions","together","pass",
// "the","output","of","first","function","call","as",
// "input","second"]
```

我们把 `words(..)` 输出的数组命名为 `wordsFound`。`unique(..)` 的输入也是一个数组，因此我们可以将 `wordsFound` 传入给它。

让我们重新回到糖果工厂的流水线：第一台机器接收的“输入”是融化的巧克力，它的“输出”是一堆成型且冷却的巧克力。流水线上的下一个机器将这堆巧克力作为它的“输入”，它的“输出”是一片片切好的巧克力糖果。下一步就是，流水线上的另一台机器将这些传送带上的小片巧克力糖果处理，并输出成包装好的糖果，准备打包和运输。



糖果工厂靠这套流程运营的很成功，但是和所有的商业公司一样，管理者们需要不停的寻找增长点。

为了跟上更多糖果的生产需求，他们决定拿掉传送带这么个玩意，直接把三台机器叠在一起，这样第一台的输出阀就直接和下一台的输入阀直接连一起了。这样第一台机器和第二台机器之间，就再也不会有一堆巧克力在传送带上慢吞吞的移动了，并且也不会有空间浪费和隆隆的噪音声了。

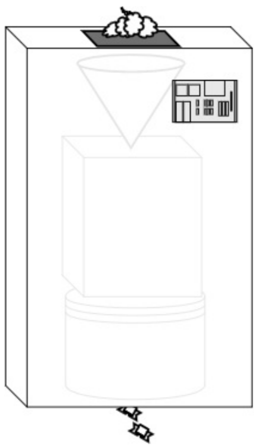
这项革新为工厂节省了很大的空间，所以管理者很高兴，他们每天能够造更多的糖果了！

等价于这种升级后的糖果工厂配置的代码跳过了中间步骤（上面代码片段中的 `wordsFound` 变量），仅仅是将两个函数调用一起使用：

```
var wordsUsed = unique( words( text ) );
```

注意：尽管我们通常以从左往右的方式阅读函数调用 ———— 先 `unique(..)` 然后 `words(..)` ———— 这里的操作顺序实际上是从右往左的，或者说是自内而外。`words(..)` 将会首先运行，然后才是 `unique(..)`。晚点我们会讨论符合我们自然的、从左往右阅读执行顺序的模式，叫做 `pipe(..)`。

堆在一起的机器工作的还不错，但有些笨重了，电线挂的到处都是。创造的机器堆越多，工厂车间就会变得越凌乱。而且，装配和维护这些机器堆太占用时间了。



有一天早上，一个糖果工厂的工程师突然想到了一个好点子。她想，如果她能在外面做一个大盒子把所有的电线都藏起来，效果肯定超级棒；盒子里面，三台机器相互连接，而盒子外面，一切都变得很整洁、干净。在这个很赞的机器的顶部，是倾倒融化巧克力的管道，在它的底部，是吐出包装好的巧克力糖果的管道。

这样一个单个的组合版机器，变得更易移动和安装到工厂需要的地方中去了。工厂的车间工人也会变得更高兴，因为他们不用再摆弄三台机子上的那些按钮和表盘了；他们很快更喜欢使用这个独立的很赞的机器。

回到代码上：我们现在了解到 `words(..)` 和 `unique(..)` 执行的特定顺序 -- 思考：组合的乐高 -- 是一种我们在应用中其它部分也能够用到的东西。所以，现在让我们定义一个组合这些玩意的函数：

```
function uniqueWords(str) {
    return unique( words( str ) );
}
```

`uniqueWords(..)` 接收一个字符串并返回一个数组。它是 `unique(..)` 和 `words(..)` 的组合，并且满足我们的数据流向要求：

```
wordsUsed <-- unique <-- words <-- text
```

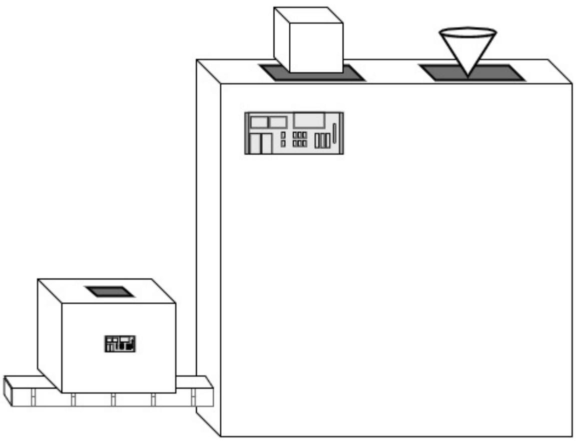
你现在应该能够明白了：糖果工厂设计模式的演变革命就是函数的组合。

制造机器

糖果工厂一切运转良好，多亏了省下的空间，他们现在有足够多的地方来尝试制作新的糖果了。鉴于之前的成功，管理者迫切的想要发明新的棒棒的组合版机器，从而制造越来越多种类的糖果。

但工厂的工程师们跟不上老板的节奏，因为每次造一台新的棒棒的组合版机器，他们就要花费很多的时间来造新的外壳，从而适应那些独立的机器。

所以工程师们联系了一家工业机器制供应商来帮他们。他们很惊讶的发现这家供应商竟然提供 机器制造 器！听起来好像不可思议，他们买入了一台这样的机器，这台机器能够将工厂中小一点的机器 —— 比如负责巧克力冷却、切割的机器 —— 自动连线，甚至在外面还自动包了一个干净的大盒子。这么牛的机器简直能把这家糖果工厂送上天了！



回到代码上，让我们定义一个实用函数叫做 `compose2(..)`，它能够自动创建两个函数的组合，这和我们手动做的是一模一样的。

```js
function compose2(fn2,fn1) {
    return function composed(origValue){
        return fn2( fn1( origValue ) );
    };
}

// ES6 箭头函数形式写法
var compose2 =
    (fn2,fn1) =>
        origValue =>
            fn2( fn1( origValue ) );
```

你是否注意到我们定义参数的顺序是 `fn2,fn1` ，不仅如此，参数中列出的第二个函数（也被称作 `fn1` ）会首先运行，然后才是参数中的第一个函数（ `fn2` ）？换句话说，这些函数是以从右往左的顺序组合的。

这看起来是种奇怪的实现，但这是有原因的。大部分传统的 FP 库为了顺序而将它们的 `compose(..)` 定义为从右往左的工作，所以我们沿袭了这种惯例。

但是为什么这么做？我认为最简单的解释（但不一定符合真实的历史）就是我们在以手动执行的书写顺序来列出它们时，或是与我们从左往右阅读这个列表时看到它们的顺序相符合。

`unique(words(str))` 以从左往右的顺序列出了 `unique, words` 函数，所以我们让 `compose2(..)` 实用函数也以这种顺序接收它们。现在，更高效的糖果制造机定义如下：

```js
var uniqueWords = compose2( unique, words );
```

组合的变体

看起来貌似 `<-- unique <-- words` 的组合方式是这两种函数能够被组合起来的唯一顺序。但我们实际上能够以另外的目的创建一个实用函数，将它们以相反的顺序组合起来。

```js
var letters = compose2( words, unique );

var chars = letters( "How are you Henry?" );
chars;
// ["h","o","w","a","r","e","y","u","n"]
```

因为 `words(..)` 实用函数，上面的代码才能正常工作。为了值类型的安全，首先使用 `String(..)` 将它的输入强转为一个字符串。所以 `unique(..)` 返回的数组 -- 现在是 `words(..)` 的输入 -- 成为了 `"H,o,w, ,a,r,e,y,u,n,?"` 这样的字符串。然后 `words(..)` 中的行为将字符串处理成为 `chars` 数组。

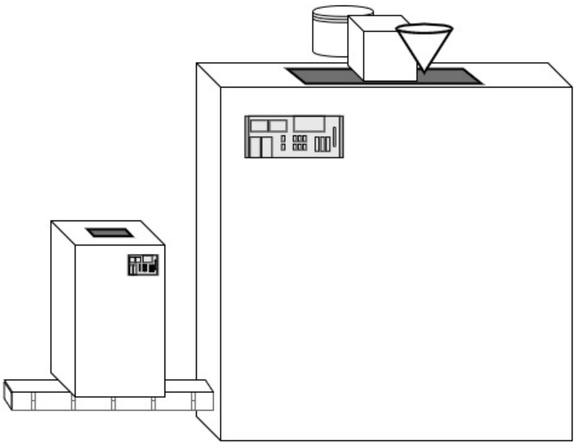
不得不承认，这是个刻意的例子。但重点是，函数的组合不总是单向的。有时候我们将灰方块放到蓝方块上，有时我们又会将蓝方块放到最上面。

假如糖果工厂尝试将包装好的糖果放入搅拌和冷却巧克力的机器，那他们最好要小心点了。

通用组合

如果我们能够定义两个函数的组合，我们也同样能够支持组合任意数量的函数。任意数目函数的组合的通用可视化数据流如下：

```
finalValue <-- func1 <-- func2 <-- ... <-- funcN <-- origValue
```



现在糖果工厂拥有了最好的制造机：它能够接收任意数量独立的小机器，并吐出一个大只的、超赞的机器，能把每一步都按照顺序做好。这个糖果制作流程简直棒呆了！简直是威利·旺卡（译者注：《查理和巧克力工厂》中的人物，他拥有一座巧克力工厂）的梦想！

我们能够像这样实现一个通用 `compose(..)` 实用函数：

```
function compose(...fns) {
    return function composed(result){
        // 拷贝一份保存函数的数组
        var list = fns.slice();

        while (list.length > 0) {
            // 将最后一个函数从列表尾部拿出
            // 并执行它
            result = list.pop()( result );
        }

        return result;
    };
}

// ES6 箭头函数形式写法
var compose =
    (...fns) =>
        result => {
            var list = fns.slice();

            while (list.length > 0) {
                // 将最后一个函数从列表尾部拿出
                // 并执行它
                result = list.pop()( result );
            }

            return result;
        };
```

现在看一下组合超过两个函数的例子。回想下我们的 `uniqueWords(..)` 组合例子，让我们增加一个 `skipShortWords(..)` 。

```
function skipShortWords(list) {
    var filteredList = [];

    for (let i = 0; i < list.length; i++) {
        if (list[i].length > 4) {
            filteredList.push( list[i] );
        }
    }

    return filteredList;
}
```

让我们再定义一个 `biggerWords(..)` 来包含 `skipShortWords(..)` 。我们期望等价的手工组合方式是 `skipShortWords(unique(words(text)))` ，所以让我们采用 `compose(..)` 来实现它：

```
var text = "To compose two functions together, pass the \
output of the first function call as the input of the \
second function call.";

var biggerWords = compose( skipShortWords, unique, words );

var wordsUsed = biggerWords( text );

wordsUsed;
// ["compose","functions","together","output","first",
// "function","input","second"]
```

现在，让我们回忆一下第 3 章中出现的 `partialRight(..)` 来让组合变的更有趣。我们能够构造一个由 `compose(..)` 自身组成的右偏函数应用，通过提前定义好第二和第三参数（`unique(..)` 和 `words(..)`）；我们把它称作 `filterWords(..)`（如下）。

然后，我们能够通过多次调用 `filterWords(..)` 来完成组合，但是每次的第一参数却各不相同。

```
// 注意： 使用 a <= 4 来检查，而不是 skipShortWords(..) 中用到的 > 4
function skipLongWords(list) { /* .. */ }

var filterWords = partialRight( compose, unique, words );

var biggerWords = filterWords( skipShortWords );
var shorterWords = filterWords( skipLongWords );

biggerWords( text );
// ["compose","functions","together","output","first",
// "function","input","second"]

shorterWords( text );
// ["to","two","pass","the","of","call","as"]
```

花些时间考虑一下基于 `compose(..)` 的右偏函数应用给了我们什么。它允许我们在组合的第一步之前做指定，然后以不同后期步骤 (`biggerWords(..)` and `shorterWords(..)`) 的组合来创建特定的变体。这是函数式编程中最强大的手段之一。

你也能通过 `curry(..)` 创建的组合来替代偏函数应用，但因为从右往左的顺序，比起只使用 `curry( compose, ..)`，你可能更想使用 `curry( reverseArgs(compose), ..)`。

注意：因为 `curry(..)`（至少我们在第 3 章中实现的是这样）依赖于探测参数数目（`length`）或手动指定其数目，而 `compose(..)` 是一个可变的函数，所以你需要手动指定数目，就像这样 `curry(.. , 3)`。

不同的实现

当然，你可能永远不会在生产中使用自己写的 `compose(..)`，而更倾向于使用某个库所提供的方案。但我发现了解底层工作的原理实际上对强化理解函数式编程中通用概念非常有用。

所以让我们看看对于 `compose(..)` 的不同实现方案。我们能看到每一种实现的优缺点，特别是性能方面。

我们将稍后在文中查看 `reduce(..)` 实用函数的细节，但现在，只需了解它将一个列表（数组）简化为一个单一的有限值。看起来像是一个很棒的循环体。

举个例子，如果在数字列表 `[1,2,3,4,5,6]` 上做加法约减，你将要循环它们，并且随着循环将它们加在一起。这一过程将首先将 `1` 加 `2`，然后将结果加 `3`，然后加 `4`，等等。最后得到总和：`21`。

原始版本的 `compose(..)` 使用一个循环并且饥渴的（也就是，立刻）执行计算，将一个调用的结果传递到下一个调用。我们可以通过 `reduce(..)` （代替循环）做到同样的事。

```
function compose(...fns) {
    return function composed(result){
        return fns.reverse().reduce( function reducer(result,fn){
            return fn( result );
        }, result );
    };
}

// ES6 箭头函数形式写法
var compose = (...fns) =>
    result =>
        fns.reverse().reduce(
            (result,fn) =>
                fn( result )
            , result
        );
```

注意到 `reduce(..)` 循环发生在最后的 `composed(..)` 运行时，并且每一个中间的 `result(..)` 将会在下一次调用时作为输入值传递给下一个迭代。

这种实现的优点就是代码更简练，并且使用了常见的函数式编程结构：`reduce(..)`。这种实现方式的性能和原始的 `for` 循环版本很相近。

但是，这种实现局限处在于外层的组合函数（也就是，组合中的第一个函数）只能接收一个参数。其他大多数实现在首次调用的时候就把所有参数传进去了。如果组合中的每一个函数都是一元的，这个方案没啥大问题。但如果你需要给第一个调用传递多参数，那么你可能需要不同的实现方案。

为了修正第一次调用的单参数限制，我们可以仍使用 `reduce(..)` ，但加一个懒执行函数包裹器：

```
function compose(...fns) {
    return fns.reverse().reduce( function reducer(fn1,fn2){
        return function composed(...args){
            return fn2( fn1( ...args ) );
        };
    } );
}

// ES6 箭头函数形式写法
var compose =
    (...fns) =>
        fns.reverse().reduce( (fn1,fn2) =>
            (...args) =>
                fn2( fn1( ...args ) )
        );
```

注意到我们直接返回了 `reduce(..)` 调用的结果，该结果自身就是个函数，不是一个计算过的值。该函数让我们能够传入任意数目的参数，在整个组合过程中，将这些参数传入到第一个函数调用中，然后依次产出结果给到后面的调用。

相较于直接计算结果并把它传入到 `reduce(..)` 循环中进行处理，这种实现通过在组合之前只运行 一次 `reduce(..)` 循环，然后将所有的函数调用运算全部延迟了 ———— 称为惰性运算。每一个简化后的局部结果都是一个包裹层级更多的函数。

当你调用最终组合函数并且提供一个或多个参数的时候，这个层层嵌套的大函数内部的所有层级，由内而外调用，以相反的方式连续执行（不是通过循环）。

这个版本的性能特征和之前 `reduce(..)` 基础实现版有潜在的差异。在这儿，`reduce(..)` 只在生成大个的组合函数时运行过一次，然后这个组合函数只是简单的一层层执行它内部所嵌套的函数。在前一版本中，`reduce(..)` 将在每一次调用中运行。

在考虑哪一种实现更好时，你的情况可能会不一样，但是要记得后面的实现方式并没有像前一种限制只能传一个参数。

我们也能够使用递归来定义 `compose(..)`。递归式定义的 `compose(fn1,fn2, .. fnN)` 看起来会是这样：

```
compose( compose(fn1,fn2, .. fnN-1), fnN );
```

注意： 我们将在第 9 章揭示更多的细节，所以如果这块看起来让你疑惑，那么暂时跳过该部分是没问题的，你可以在阅读完第 9 章后再来看。

这里是我们用递归实现 `compose(..)` 的代码：

```
function compose(...fns) {
    // 拿出最后两个参数
    var [ fn1, fn2, ...rest ] = fns.reverse();

    var composedFn = function composed(...args){
        return fn2( fn1( ...args ) );
    };

    if (rest.length == 0) return composedFn;

    return compose( ...rest.reverse(), composedFn );
}

// ES6 箭头函数形式写法
var compose =
    (...fns) => {
        // 拿出最后两个参数
        var [ fn1, fn2, ...rest ] = fns.reverse();

        var composedFn =
            (...args) =>
                fn2( fn1( ...args ) );

        if (rest.length == 0) return composedFn;

        return compose( ...rest.reverse(), composedFn );
    };
```

我认为递归实现的好处是更加概念化。我个人觉得相较于不得不在循环里跟踪运行结果，通过递归的方式进行重复的动作反而更易懂。所以我更喜欢以这种方式的代码来表达。

其他人可能会觉得递归的方法在智力上造成的困扰更让人有些畏惧。我建议你作出自己的评估。

重排序组合

我们早期谈及的是从右往左顺序的标准 `compose(..)` 实现。这么做的好处是能够和手工组合列出参数（函数）的顺序保持一致。

不足之处就是它们排列的顺序和它们执行的顺序是相反的，这将会造成困扰。同时，不得不使用 `partialRight(compose, ..)` 提早定义要在组合过程中 第一个 执行的函数。

相反的顺序，从右往左的组合，有个常见的名字：`pipe(..)`。这个名字据说来自 Unix/Linux 界，那里大量的程序通过“管道传输”（`|` 运算符）第一个的输出到第二个的输入，等等（即，`ls -la | grep "foo" | less`）。

`pipe(..)` 与 `compose(..)` 一模一样，除了它将列表中的函数从左往右处理。

```
function pipe(...fns) {
    return function piped(result){
        var list = fns.slice();

        while (list.length > 0) {
            // 从列表中取第一个函数并执行
            result = list.shift()( result );
        }

        return result;
    };
}
```

实际上，我们只需将 `compose(..)` 的参数反转就能定义出来一个 `pipe(..)`。

```
var pipe = reverseArgs( compose );
```

非常简单！

回忆下之前的通用组合的例子：

```
var biggerWords = compose( skipShortWords, unique, words );
```

以 `pipe(..)` 的方式来实现，我们只需要反转参数的顺序：

```
var biggerWords = pipe( words, unique, skipShortWords );
```

`pipe(..)` 的优势在于它以函数执行的顺序排列参数，某些情况下能够减轻阅读者的疑惑。`pipe(words,unique,skipShortWords)` 看起来和读起来会更简单，能知道我们首先执行 `words(..)`，然后 `unique(..)`，最后是 `skipShortWords(..)`。

假如你想要部分的应用第一个函数（们）来负责执行，`pipe(..)` 同样也很方便。就像我们之前使用 `compose(..)` 构建的右偏函数应用一样。

对比：

```
var filterWords = partialRight( compose, unique, words );

// vs

var filterWords = partial( pipe, words, unique );
```

你可能会回想起第 3 章 `partialRight(..)` 中的定义，它实际使用了 `reverseArgs(..)`，就像我们的 `pipe(..)` 现在所做的。所以，不管怎样，我们得到了同样的结果。

在这一特定场景下使用 `pipe(..)` 的轻微性能优势在于我们不必再通过右偏函数应用的方式来使用 `compose(..)` 保存从右往左的参数顺序，使用 `pipe(..)` 我们不必再跟 `partialRight(..)` 一样需要将参数顺序反转回去。所以在这里 `partial(pipe, ..)` 比 `partialRight(compose, ..)` 要好一点。

一般来说，在使用一个完善的函数式编程库时，`pipe(..)` 和 `compose(..)` 没有明显的性能区别。

抽象

抽象经常被定义为对两个或多个任务公共部分的剥离。通用部分只定义一次，从而避免重复。为了展现每个任务的特殊部分，通用部分需要被参数化。

举个例子，思考如下（明显刻意生成的）代码：

```
function saveComment(txt) {
    if (txt != "") {
        comments[comments.length] = txt;
    }
}

function trackEvent(evt) {
    if (evt.name !== undefined) {
        events[evt.name] = evt;
    }
}
```

这两个实用函数都是将一个值存入一个数据源，这是通用的部分。不同的是一个是将值放置到数组的末尾，另一个是将值放置到对象的某个属性上。

让我们抽象一下：

```
function storeData(store,location,value) {
    store[location] = value;
}

function saveComment(txt) {
    if (txt != "") {
        storeData( comments, comments.length, txt );
    }
}

function trackEvent(evt) {
    if (evt.name !== undefined) {
        storeData( events, evt.name, evt );
    }
}
```

引用一个对象（或数组，多亏了 JS 中方便的 `[]` 符号）属性和将值设入的通用任务被抽象到独立的 `storeData(..)` 函数。这个函数当前只有一行代码，该函数能提出其它多任务中通用的行为，比如生成唯一的数字 ID 或将时间戳存入。

如果我们在多处重复通用的行为，我们将会面临改了几处但忘了改别处的维护风险。在做这类抽象时，有一个原则是，通常被称作 DRY（don't repeat yourself）。

DRY 力求能在程序的任何任务中有唯一的定义。代码不够 DRY 的另一个托辞就是程序员们太懒，不想做非必要的工作。

抽象能够走得更远。思考：

```
function conditionallyStoreData(store,location,value,checkFn) {
    if (checkFn( value, store, location )) {
        store[location] = value;
    }
}

function notEmpty(val) { return val != ""; }

function isUndefined(val) { return val === undefined; }

function isPropUndefined(val,obj,prop) {
    return isUndefined( obj[prop] );
}

function saveComment(txt) {
    conditionallyStoreData( comments, comments.length, txt, notEmpty );
}

function trackEvent(evt) {
    conditionallyStoreData( events, evt.name, evt, isPropUndefined );
}
```

为了实现 DRY 和避免重复的 `if` 语句，我们将条件判断移动到了通用抽象中。我们同样假设在程序中其它地方可能会检查非空字符串或非 `undefined` 的值，所以我们也能将这些东西 DRY 出来。

这些代码现在变得更 DRY 了，但有些抽象过度了。开发者需要对他们程序中每个部分使用恰当的抽象级别保持谨慎，不能太过，也不能不够。

关于我们在本章中对函数的组合进行的大量讨论，看起来它的好处是实现这种 DRY 抽象。但让我们别急着下结论，因为我认为组合实际上在我们的代码中发挥着更重要的作用。

而且，即使某些东西只出现了一次，组合仍然十分有用（没有重复的东西可以被抽出来）。

除了通用化和特殊化的对比，我认为抽象有更多有用的定义，正如下面这段引用所说：

> ... 抽象是一个过程，程序员将一个名字与潜在的复杂程序片段关联起来，这样该名字就能够被认为代表函数的目的，而不是代表函数如何实现的。通过隐藏无关的细节，抽象降低了概念复杂度，让程序员在任意时间都可以集中注意力在程序内容中的可维护子集上。
>
> 《程序设计语言》， 迈克尔 L 斯科特
>
> https://books.google.com/books?id=jM-cBAAAQBAJ&pg=PA115&lpg=PA115&dq=%22making+it+possible+for+the+programmer+to+focus+on+a+manageable+subset%22&source=bl&ots=yrJ3a-Tvi6&sig=XZwYoWwbQxP2w5qh2k2uMAPj47k&hl=en&sa=X&ved=0ahUKEwjKr-Ty35DSAhUJ4mMKHbPrAUUQ6AEIIzAA#v=onepage&q=%22making%20it%20possible%20for%20the%20programmer%20to%20focus%20on%20a%20manageable%20subset%22&f=false

// TODO: 给这本书或引用弄一个更好的参照，至少找到一个更好的在线链接

这段引用表述的观点是抽象 ———— 通常来说，是指把一些代码片段放到自己的函数中 ———— 是围绕着能将两部分功能分离，从而达到可以专注于某一独立的部分为主要目的来服务的。

需要注意的是，这种场景下的抽象并不是为了隐藏细节，比如把一些东西当作黑盒来对待。这一观念其实更贴近于编程中的封装性原则。我们不是为了隐藏细节而抽象，而是为了通过分离来突出关注点。

还记得这段文章的开头，我说函数式编程的目的是为了创造更可读、更易理解的代码。一个有效的方法是将交织缠绕的 ———— 紧紧编织在一起，像一股绳子 ———— 代码解绑为分离的、更简单的 ———— 松散绑定的 ———— 代码片段。以这种方式来做的话，代码的阅读者将不会在寻找其它部分细节的时候被其中某块的细节所分心。

我们更高的目标是不只对某些东西实现一次，这是 DRY 的观念。实际上，有些时候我们确实在代码中不断重复。于是，我们寻求更分离的实现方式。我们尝试突出关注点，因为这能提高可读性。

另一种描述这个目标的方式就是 ———— 通过命令式 vs 声明式的编程风格。命令式代码主要关心的是描述怎么做来准确完成一项任务。声明式代码则是描述输出应该是什么，并将具体实现交给其它部分。

换句话说，声明式代码从怎么做中抽象出了是什么。尽管普通的声明式代码在可读性上强于命令式，但没有程序（除了机器码 1 和 0）是完全的声明式或者命令式代码。编程者必须在它们之间寻找平衡。

ES6 增加了很多语法功能，能将老的命令式操作转换为新的声明式形式。可能最清晰的当属解构了。解构是一种赋值模式，它描述了如何将组合值（对象、数组）内的构成值分解出来的方法。

这里是一个数组解构的例子：

```
function getData() {
    return [1,2,3,4,5];
}

// 命令式
var tmp = getData();
var a = tmp[0];
var b = tmp[3];

// 声明式
var [ a ,,, b ] = getData();
```

是什么就是将数组中的第一个值赋给 `a`，然后第四个值赋给 `b`。怎么做就是得到一个数组的引用（`tmp`）然后手动的通过数组索引 `0` 和 `3`，分别赋值给 `a` 和 `b`。

数组的解构是否隐藏了赋值细节？这要看你看待的角度了。我认为它知识简单的将是什么从怎么做中分离出来。JS 引擎仍然做了赋值的工作，但它阻止了你自己去抽象怎么做的过程。

相反的是，你阅读 `[ a ,,, b ] = ..` 的时候，便能看到该赋值模式只不过是告诉你将要发生的是什么。数组的解构是声明式抽象的一个例子。

将组合当作抽象

函数组合到底做了什么？函数组合同样也是一种声明式抽象。

回想下之前的 `shorterWords(..)` 例子。让我们对比下命令式和声明式的定义。

```
// 命令式
function shorterWords(text) {
    return skipLongWords( unique( words( text ) ) );
}

// 声明式
var shorterWords = compose( skipLongWords, unique, words );
```

声明式关注点在是什么上 -- 这 3 个函数传递的数据从一个字符串到一系列更短的单词 -- 并且将怎么做留在了 `compose(..)` 的内部。

在一个更大的层面上看，`shorterWords = compose(..)` 行解释了怎么做来定义一个 `shorterWords(..)` 实用函数，这样在代码的别处使用时，只需关注下面这行声明式的代码输出是什么。

```
shorterWords( text );
```

组合将一步步得到一系列更短的单词的过程抽象了出来。

相反的看，如果我们不使用组合抽象呢？

```
var wordsFound = words( text );
var uniqueWordsFound = unique( wordsFound );
skipLongWords( uniqueWordsFound );
```

或者这种：

```
skipLongWords( unique( words( text ) ) );
```

这两个版本展示的都是一种更加命令式的风格，违背了声明式风格优先原则。阅读者关注这两个代码片段时，会被更多的要求了解怎么做而不是是什么。

函数组合并不是通过 DRY 的原则来节省代码量。即使 `shorterWords(..)` 的使用只出现了一次 -- 所以并没有重复问题需要避免！-- 从怎么做中分离出是什么仍能帮助我们提升代码。

组合是一个抽象的强力工具，它能够将命令式代码抽象为更可读的声明式代码。

回顾形参

既然我们已经把组合都了解了一遍 -- 那么是时候抛出函数式编程中很多地方都有用的小技巧了 -- 让我们通过在某个场景下回顾第 3 章的“无形参”（译者注：“无形参”指的是移除对函数形参的引用）段落中的 `point-free` 代码，并把它重构的稍微复杂点来观察这种小技巧。

```
// 提供该API：ajax( url, data, cb )
var getPerson = partial( ajax, "http://some.api/person" );
var getLastOrder = partial( ajax, "http://some.api/order", { id: -1 } );

getLastOrder( function orderFound(order){
    getPerson( { id: order.personId }, function personFound(person){
        output( person.name );
    } );
} );
```

我们想要移除的“点”是对 `order` 和 `person` 参数的引用。

让我们尝试将 `person` 形参移出 `personFound(..)` 函数。要达到目的，我们需要首先定义：

```
function extractName(person) {
    return person.name;
}
```

但据我们观察这段操作能够表达的更通用些：将任意对象的任意属性通过属性名提取出来。让我们把这个实用函数称为 `prop(..)`：

```
function prop(name,obj) {
    return obj[name];
}

// ES6 箭头函数形式
var prop =
    (name,obj) =>
        obj[name];
```

我们处理对象属性的时候，也需要定义下反操作的工具函数：`setProp(..)`，为了将属性值设到某个对象上。

但是，我们想小心一些，不改动现存的对象，而是创建一个携带变化的复制对象，并将它返回出去。这样处理的原因将在第 5 章中讨论更多细节。

```
function setProp(name,obj,val) {
    var o = Object.assign( {}, obj );
    o[name] = val;
    return o;
}
```

现在，定义一个 `extractName(..)`，它能将对象中的 `"name"` 属性拿出来，我们将部分应用 `prop(..)`：

```
var extractName = partial( prop, "name" );
```

注意：不要误解这里的 `extractName(..)`，它其实什么都还没有做。我们只是部分应用 `prop(..)` 来创建了一个等待接收包含 `"name"` 属性的对象的函数。我们也能通过 `curry(prop)("name")` 做到一样的事。

下一步，让我们缩小关注点，看下例子中嵌套的这块查找操作的调用：

```
getLastOrder( function orderFound(order){
    getPerson( { id: order.personId }, outputPersonName );
} );
```

我们该如何定义 `outputPersonName(..)`？为了方便形象化我们所需要的东西，想一下我们需要的数据流是什么样：

```
output <-- extractName <-- person
```

`outputPersonName(..)` 需要是一个接收（对象）值的函数，并将它传递给 `extractName(..)`，然后将处理后的值传给 `output(..)`。

希望你能看出这里需要 `compose(..)` 操作。所以我们能够将 `outputPersonName(..)` 定义为：

```
var outputPersonName = compose( output, extractName );
```

我们刚刚创建的 `outputPersonName(..)` 函数是提供给 `getPerson(..)` 的回调。所以我们还能定义一个函数叫做 `processPerson(..)` 来处理回调参数，使用 `partialRight(..)`：

```
var processPerson = partialRight( getPerson, outputPersonName );
```

让我们用新函数来重构下之前的代码：

```
getLastOrder( function orderFound(order){
    processPerson( { id: order.personId } );
} );
```

唔，进展还不错！

但我们需要继续移除掉 `order` 这个“形参”。下一步是观察 `personId` 能够被 `prop(..)` 从一个对象（比如 `order`）中提取出来，就像我们在 `person` 对象中提取 `name` 一样。

```
var extractPersonId = partial( prop, "personId" );
```

为了创建传递给 `processPerson(..)` 的对象（ `{ id: .. }` 的形式），让我们创建一个实用函数 `makeObjProp(..)` ，用来以特定的属性名将值包装为一个对象。

```
function makeObjProp(name,value) {
    return setProp( name, {}, value );
}

// ES6 箭头函数形式
var makeObjProp =
    (name,value) =>
        setProp( name, {}, value );
```

提示： 这个实用函数在 Ramda 库中被称为 `objOf(..)` 。

就像我们之前使用 `prop(..)` 来创建 `extractName(..)` ，我们将部分应用 `makeObjProp(..)` 来创建 `personData(..)` 函数用来制作我们的数据对象。

```
var personData = partial( makeObjProp, "id" );
```

为了使用 `processPerson(..)` 来完成通过 `order` 值查找一个人的功能，我们需要的数据流如下：

```
processPerson <-- personData <-- extractPersonId <-- order
```

所以我们只需要再使用一次 `compose(..)` 来定义一个 `lookupPerson(..)` ：

```
var lookupPerson = compose( processPerson, personData, extractPersonId );
```

然后，就是这样了！把这整个例子重新组合起来，不带任何的“形参”：

```
var getPerson = partial( ajax, "http://some.api/person" );
var getLastOrder = partial( ajax, "http://some.api/order", { id: -1 } );

var extractName = partial( prop, "name" );
var outputPersonName = compose( output, extractName );
var processPerson = partialRight( getPerson, outputPersonName );
var personData = partial( makeObjProp, "id" );
var extractPersonId = partial( prop, "personId" );
var lookupPerson = compose( processPerson, personData, extractPersonId );

getLastOrder( lookupPerson );
```

哇哦。没有形参。并且 `compose(..)` 在两处地方看起来相当有用！

我认为在这样的场景下，即使推导出我们最终答案的步骤有些多，但最终的代码却变得更加可读，因为我们不用再去详细的调用每一步了。

即使你不想看到或命名这么多中间步骤，你依然可以通过不使用独立变量而是将表达式串起来来保留无点特性。

```
partial( ajax, "http://some.api/order", { id: -1 } )
(
    compose(
        partialRight(
            partial( ajax, "http://some.api/person" ),
            compose( output, partial( prop, "name" ) )
        ),
        partial( makeObjProp, "id" ),
        partial( prop, "personId" )
    )
);
```

这段代码肯定没那么罗嗦了，但我认为比之前的每个操作都有其对应的变量相比，可读性略有降低。但是不管怎样，组合帮助我们实现了无点的风格。

总结

函数组合是一种定义函数的模式，它能将一个函数调用的输出路由到另一个函数的调用上，然后一直进行下去。

因为 JS 函数只能返回单个值，这个模式本质上要求所有组合中的函数（可能第一个调用的函数除外）是一元的，当前函数从上一个函数输出中只接收一个输入。

相较于在我们的代码里详细列出每个调用，函数组合使用 `compose(..)` 实用函数来提取出实现细节，让代码变得更可读，让我们更关注组合完成的是什么，而不是它具体做什么。

组合 ———— 声明式数据流 ———— 是支撑函数式编程其他特性的最重要的工具之一。

JavaScript 轻量级函数式编程

第 5 章：减少副作用

在第 2 章，我们讨论了一个函数除了它的返回值之外还有什么输出。现在你应该很熟悉用函数式编程的方法定义一个函数了，所以对于函数式编程的副作用你应该有所了解。

我们将检查各种各样不同的副作用并且要看看他们为什么会对我们的代码质量和可读性造成损害。

这一章的要点是：编写出没有副作用的程序是不可能的。当然，也不是不可能，你当然可以编写出没有副作用的程序。但是这样的话程序就不会做任何有用和明显的事情。如果你编写出来一个零副作用的程序，你就无法区分它和一个被删除的或者空程序的区别。

函数式编程者并没有消除所有的副作用。实际上，我们的目标是尽可能地限制他们。要做到这一点，我们首先需要完全理解函数式编程的副作用。

什么是副作用

因果关系：举一个我们人类对周围世界影响的最基本、最直观的例子，推一下放在桌子边沿上的一本书，书会掉落。不需要你拥有一个物理学的学位你也会知道，这是因为你刚刚推了书并且书掉落是因为地心引力，这是一个明确并直接的关系。

在编程中，我们也完全会处理因果关系。如果你调用了一个函数（起因），就会在屏幕上输出一条消息（结果）。

当我们在阅读程序的时候，能够清晰明确的识别每一个起因和每一个结果是非常重要的。在某种程度上，通读程序但不能看到因果的直接关系，程序的可读性就会降低。

思考一下：

```
function foo(x) {
    return x * 2;
}

var y = foo( 3 );
```

在这段代码中，有很直接的因果关系，调用值为 `3` 的 foo 将具有返回值 `6` 的效果，调用函数 foo() 是起因，然后将其赋值给 `y` 是结果。这里没有歧义，传入参数为 3 将会返回 6，将函数结果赋值给变量 `y` 是结果。

但是现在：

```
function foo(x) {
    y = x * 2;
}

var y;

foo( 3 );
```

这段代码有相同的输出，但是却有很大的差异，这里的因果是没有联系的。这个影响是间接的。这种方式设置 `y` 就是我们所说的副作用。

注意：当函数引用外部变量时，这个变量就称为自由变量。并不是所有的自由变量引用都是不好的，但是我们要对它们非常小心。

假使给你一个引用来调用函数 `bar(..)`，你看不到代码，但是我告诉你这段代码并没有间接的副作用，只有一个显式的 `return` 值会怎么样？

```
bar( 4 );            // 42
```

因为你知道 `bar(..)` 的内部结构不会有副作用，你可以像这样直接地调用 `bar(..)`。但是如果你不知道 `bar(..)` 没有副作用，为了理解调用这个函数的结果，你必须去阅读和分析它的逻辑。这对读者来说是额外的负担。

有副作用的函数可读性更低，因为它需要更多的阅读来理解程序。

但是程序往往比这个要复杂，思考一下：

```
var x = 1;

foo();

console.log( x );

bar();

console.log( x );

baz();

console.log( x );
```

你能确定每次 `console.log(x)` 的值都是你想要的吗？

答案是否定的。如果你不确定函数 `foo()` 、 `bar()` 和 `baz()` 是否有副作用，你就不能保证每一步的 `x` 将会是什么，除非你检查每个步骤的实现，然后从第一行开始跟踪程序，跟踪所有状态的改变。

换句话说， `console.log(x)` 最后的结果是不能分析和预测的，除非你已经在心里将整个程序执行到这里了。

猜猜谁擅长运行你的程序？JS 引擎。猜猜谁不擅长运行你的程序？你代码的读者。然而，如果你选择在一个或多个函数调用中编写带有（潜在）副作用的代码，那么这意味着你已经使你的读者必须将你的程序完整地执行到某一行，以便他们理解这一行。

如果 `foo()` 、 `bar()` 、和 `baz()` 都没有副作用的话，它们就不会影响到 `x` ，这就意味着我们不需要在心里默默地执行它们并且跟踪 `x` 的变化。这在精力上负担更小，并且使得代码更加地可读。

潜在的原因

输出和状态的变化，是最常被引用的副作用的表现。但是另一个有损可读性的实践是一些被认为的侧因，思考一下：

```
function foo(x) {
    return x + y;
}

var y = 3;

foo( 1 );           // 4
```

`y` 不会随着 `foo(..)` 改变，所以这和我们之前看到的副作用有所不同。但是现在，对函数 `foo(..)` 的调用实际上取决于 `y` 当前的状态。之后我们如果这样做：

```
y = 5;

// ..

foo( 1 );           // 6
```

我们可能会感到惊讶两次调用 `foo(1)` 返回的结果不一样。

`foo(..)` 对可读性有一个间接的破坏性。如果没有对函数 `foo(..)` 进行仔细检查，使用者可能不会知道导致这个输出的原因。这看起来仅仅像是参数 `1` 的原因，但却不是这样的。

为了帮助可读性，所有决定 `foo(..)` 输出的原因应该被设置的直接并明显。函数的使用者将会直接看到原因和结果。

使用固定的状态

避免副作用就意味着函数 `foo(..)` 不能引用自由变量了吗？

思考下这段代码：

```
function foo(x) {
    return x + bar( x );
}

function bar(x) {
    return x * 2;
}

foo( 3 );            // 9
```

很明显，对于函数 `foo(..)` 和函数 `bar(..)`，唯一和直接的原因就是参数 `x`。但是 `bar(x)` 被称为什么呢？`bar` 仅仅只是一个标识符，在 JS 中，默认情况下，它甚至不是一个常量（不可重新分配的变量）。`foo(..)` 函数依赖于 `bar` 的值，`bar` 作为一个自由变量被第二个函数引用。

所以说这个函数还依赖于其他的原因吗？

我认为不。虽然可以用其他的函数来重写 `bar` 这个变量，但是在代码中我没有这样做，这也不是我的惯例或先例。无论出于什么意图和目的，我的函数都是常量（从不重新分配）。

思考一下：

```
const PI = 3.141592;

function foo(x) {
    return x * PI;
}

foo( 3 );            // 9.424776000000001
```

注意： JavaScript 有内置的 `Math.PI` 属性，所以我们在本文中仅仅是用 `PI` 做一个方便的说明。在实践中，总是使用 `Math.PI` 而不是你自己定义的。

上面的代码怎么样呢？`PI` 是函数 `foo(..)` 的一个副作用吗？

两个观察结果将会合理地帮助我们回答这个问题：

1. 想一下是否每次调用 `foo(3)`，都将会返回 `9.424..`？答案是肯定的。 如果每一次都给一个相同的输入（`x`），那么都将会返回相同的输出。

2. 你能用 `PI` 的当前值来代替每一个 `PI` 吗，并且程序能够和之前一样正确地的运行吗？是的。程序没有任何一部分依赖于 `PI` 值的改变，因为 `PI` 的类型是 `const`，它是不能再分配的，所以变量 `PI` 在这里只是为了便于阅读和维护。它的值可以在不改变程序行为的情况下内联。

我的结论是：这里的 `PI` 并不违反减少或避免副作用的精神。在之前的代码也没有调用 `bar(x)`。

在这两种情况下，`PI` 和 `bar` 都不是程序状态的一部分。它们是固定的，不可重新分配的（“常量”）的引用。如果他们在整个程序中都不改变，那么我们就不需要担心将他们作为变化的状态追踪他们。同样的，他们不会损害程序的可读性。而且它们也不会因为变量以不可预测的方式变化，而成为错误的源头。

注意：在我看来，使用 `const` 并不能说明 `PI` 不是副作用；使用 `var PI` 也会是同样的结果。`PI` 没有被重新分配是问题的关键，而不是使用 `const`。我们将在后面的章节讨论 `const`。

随机性

你以前可能从来没有考虑过，但是随机性是不纯的。一个使用 `Math.random()` 的函数永远都不是纯的，因为你不能根据它的输入来保证和预测它的输出。所以任何生成唯一随机的 ID 等都需要依靠程序的其他原因。

在计算中，我们使用的是伪随机算法。事实证明，真正的随机是非常难的，所以我们只是用复杂的算法来模拟它，产生的值看起来是随机的。这些算法计算很长的一串数字，但秘密是，如果你知道起始点，实际上这个序列是可以预测的。这个起点被称之为种子。

一些语言允许你指定生成随机数的种子。如果你总是指定了相同的种子，那么你将始终从后续的“随机数”中得到相同的输出序列。这对于测试是非常有用的，但是在真正的应用中使用也是非常危险的。

在 JS 中，`Math.random()` 的随机性计算是基于间接输入，因为你不能明确种子。因此，我们必须将内建的随机数生成视为不纯的一方。

I/O 效果

这可能不太明显，但是最常见（并且本质上不可避免）的副作用就是 I/O（输入/输出）。一个没有 I/O 的程序是完全没有意义的，因为它的工作不能以任何方式被观察到。一个有用的程序必须最少有一个输出，并且也需要输入。输入会产生输出。

用户事件（鼠标、键盘）是 JS 编程者在浏览器中使用的典型的输入，而输出的则是 DOM。如果你使用 Node.js 比较多，你更有可能接收到和输出到文件系统、网络系统和/或者 `stdin` / `stdout`（标准输入流/标准输出流）的输入和输出。

事实上，这些来源既可以是输入也可以是输出，是因也是果。以 DOM 为例，我们更新（产生副作用的结果）一个 DOM 元素为了给用户展示文字或图片信息，但是 DOM 的当前状态是对这些操作的隐式输入（产生副作用的原因）。

其他的错误

在程序运行期间副作用可能导致的错误是多种多样的。让我们来看一个场景来说明这些危害，希望它们能帮助我们辨认出在我们自己的程序中类似的错误。

思考一下：

```
var users = {};
var userOrders = {};

function fetchUserData(userId) {
    ajax( "http://some.api/user/" + userId, function onUserData(userData){
        users[userId] = userData;
    } );
}

function fetchOrders(userId) {
    ajax( "http://some.api/orders/" + userId, function onOrders(orders){
        for (let i = 0; i < orders.length; i++) {
                // 对每个用户的最新订单保持引用
            users[userId].latestOrder = orders[i];
            userOrders[orders[i].orderId] = orders[i];
        }
    } );
}

function deleteOrder(orderId) {
    var user = users[ userOrders[orderId].userId ];
    var isLatestOrder = (userOrders[orderId] == user.latestOrder);

    // 删除用户的最新订单？
    if (isLatestOrder) {
        hideLatestOrderDisplay();
    }

    ajax( "http://some.api/delete/order/" + orderId, function onDelete(success){
        if (success) {
                // 删除用户的最新订单？
            if (isLatestOrder) {
                user.latestOrder = null;
            }

            userOrders[orderId] = null;
        }
        else if (isLatestOrder) {
            showLatestOrderDisplay();
        }
    } );
}
```

我敢打赌，一些读者显然会发现其中潜在的错误。如果回调 `onOrders(..)` 在回调 `onUserData(..)` 之前运行，它会给一个尚未设置的值（ `users[userId]` 的 `userData` 对象）添加一个 `latestOrder` 属性

因此，这种依赖于因果关系的“错误”是在两种不同操作（是否异步）紊乱情况下发生的，我们期望以确定的顺序运行，但在某些情况下，可能会以不同的顺序运行。有一些策略可以确保操作的顺序，很明显，在这种情况下顺序是至关重要的。

这里还有另一个细小的错误，你发现了吗？

思考下这个调用顺序：

```
fetchUserData( 123 );
onUserData(..);
fetchOrders( 123 );
onOrders(..);

// later

fetchOrders( 123 );
deleteOrder( 456 );
onOrders(..);
onDelete(..);
```

你发现每一对 `fetchOrders(..)` / `onOrders(..)` 和 `deleteOrder(..)` / `onDelete(..)` 都是交替出现了吗？这个潜在的排序会伴随着我们状态管理的侧因/副作用暴露出一个古怪的状态。

在设置 `isLatestOrder` 标志和使用它来决定是否应该清空 `users` 中的用户数据对象的 `latestOrder` 属性时，会有一个延迟（因为回调）。在此延迟期间，如果 `onOrders(..)` 销毁，它可以潜在地改变用户的 `latestOrder` 引用的顺序值。当 `onDelete(..)` 在销毁之后，它会假定它仍然需要重新引用 `latestOrder` 。

错误：数据（状态）可能不同步。当进入 `onOrders(..)` 时， `latestOrder` 可能仍然指向一个较新的顺序，这样 `latestOrder` 就会被重置。

这种错误最糟糕的是你不能和其他错误一样得到程序崩溃的异常。我们只是有一个不正确的状态，同时我们的应用程序“默默地”崩溃。

`fetchUserData(..)` 和 `fetchOrders(..)` 的序列依赖是相当明显的，并且被直截了当地处理。但是，在 `fetchOrders(..)` 和 `deleteOrder(..)` 之间存在潜在的序列依赖关系，就不太清楚了。这两个似乎更加独立。并且确保他们的顺序被保留是比较棘手的，因为你事先不知道（在 `fetchOrders(..)` 产生结果之前）是否必须要按照这样的顺序执行。

是的，一旦 `deleteOrder(..)` 销毁，你就能重新计算 `isLatestOrder` 标志。但是现在你有另一个问题：你的 UI 状态可能不同步。

如果你之前已经调用过 `hideLatestOrderDisplay()` ，现在你需要调用 `showLatestOrderDisplay()` ，但是如果一个新的 `latestOrder` 已经被设置好了，你将要跟踪至少三个状态：被删除的状态是否本来是“最新的”、是否是“最新”设置的，和这两个顺序有什么不同吗？这些都是可以解决的问题，但无论如何都是不明显的。

所有这些麻烦都是因为我们决定在一组共享的状态下构造出有副作用的代码。

函数式编程人员讨厌这类因果的错误，因为这有损我们的阅读、推理、验证和最终相信代码的能力。这就是为什么他们要如此严肃地对待避免副作用的原因。

有很多避免/修复副作用的策略。我们将在本章后面和后面的章节中讨论。我要说一个确定的事情：写出有副作用/效果的代码是很正常的，所以我们需要谨慎和刻意地避免产生有副作用的代码。

一次就好

如果你必须要使用副作用来改变状态，那么一种对限制潜在问题有用的操作是幂等。如果你的值的更新是幂次的，那么数据将会适应你可能有不同副作用来源的多个此类更新的情况。

幂等的定义有点让人困惑，同时数学家和程序员使用幂等的含义稍有不同。然而，这两种观点对于函数式编程人员都是有用的。

首先，让我们给出一个计数器的例子，它既不是数学上的，也不是程序上的幂等：

```
function updateCounter(obj) {
    if (obj.count < 10) {
        obj.count++;
        return true;
    }

    return false;
}
```

这个函数通过引用递增 `obj.count` 来该改变一个对象，所以对这个对象产生了副作用。当 `o.count` 小于 10 时，如果 `updateCounter(o)` 被多次调用，即程序状态每次都要更改。另外，`updateCounter(..)` 的输出是一个布尔值，这不适合返回到 `updateCounter(..)` 的后续调用。

数学中的幂等

从数学的角度来看，幂等指的是在第一次调用后，如果你将该输出一次又一次地输入到操作中，其输出永远不会改变的操作。换句话说，`foo(x)` 将产生与 `foo(foo(x))`、`foo(foo(foo(x)))` 等相同的输出。

一个典型的数学例子是 `Math.abs(..)`（取绝对值）。`Math.abs(-2)` 的结果是 `2`，和 `Math.abs(Math.abs(Math.abs(Math.abs(-2))))` 的结果相同。像 `Math.min(..)`、`Math.max(..)`、`Math.round(..)`、`Math.floor(..)` 和 `Math.ceil(..)` 这些工具函数都是幂等的。

我们可以用同样的特征来定义一些数学运算：

```
function toPower0(x) {
    return Math.pow( x, 0 );
}

function snapUp3(x) {
    return x - (x % 3) + (x % 3 > 0 && 3);
}

toPower0( 3 ) == toPower0( toPower0( 3 ) );            // true

snapUp3( 3.14 ) == snapUp3( snapUp3( 3.14 ) );        // true
```

数学上的幂等不仅限于数学运算。我们还可以用 JavaScript 的原始类型来说明幂等的另一种形式：

```
var x = 42, y = "hello";

String( x ) === String( String( x ) );                // true

Boolean( y ) === Boolean( Boolean( y ) );            // true
```

在本文的前面，我们探究了一种常见的函数式编程工具，它可以实现这种形式的幂等：

```
identity( 3 ) === identity( identity( 3 ) );    // true
```

某些字符串操作自然也是幂等的，例如：

```
function upper(x) {
    return x.toUpperCase();
}

function lower(x) {
    return x.toLowerCase();
}

var str = "Hello World";

upper( str ) == upper( upper( str ) );                // true

lower( str ) == lower( lower( str ) );                // true
```

我们甚至可以以一种幂等方式设计更复杂的字符串格式操作，比如：

```
function currency(val) {
    var num = parseFloat(
        String( val ).replace( /[^\d.-]+/g, "" )
    );
    var sign = (num < 0) ? "-" : "";
    return `${sign}$${Math.abs( num ).toFixed( 2 )}`;
}

currency( -3.1 );                                   // "-$3.10"

currency( -3.1 ) == currency( currency( -3.1 ) );    // true
```

`currency(..)` 举例说明了一个重要的技巧：在某些情况下，开发人员可以采取额外的步骤来规范化输入/输出操作，以确保操作是幂等的来避免意外的发生。

在任何可能的情况下通过幂等的操作限制副作用要比不做限制的更新要好得多。

编程中的幂等

幂等的面向程序的定义也是类似的，但不太正式。编程中的幂等仅仅是 `f(x);` 的结果与 `f(x); f(x)` 相同而不是要求 `f(x) === f(f(x))`。换句话说，之后每一次调用 `f(x)` 的结果和第一次调用 `f(x)` 的结果没有任何改变。

这种观点更符合我们对副作用的观察。因为这更像是一个 `f(..)` 创建了一个幂等的副作用而不是必须要返回一个幂等的输出值。

这种幂等性的方式经常被用于 HTTP 操作（动词），例如 GET 或 PUT。如果 HTTP REST API 正确地遵循了幂等的规范指导，那么 PUT 被定义为一个更新操作，它可以完全替换资源。同样的，客户端可以一次或多次发送 PUT 请求（使用相同的数据），而服务器无论如何都将具有相同的结果状态。

让我们用更具体的编程方法来考虑这个问题，来检查一下使用幂等和没有使用幂等是否产生副作用：

```
// 幂等的：
obj.count = 2;
a[a.length - 1] = 42;
person.name = upper( person.name );

// 非幂等的：
obj.count++;
a[a.length] = 42;
person.lastUpdated = Date.now();
```

记住：这里的幂等性的概念是每一个幂等运算（比如 `obj.count = 2`）可以重复多次，而不是在第一次更新后改变程序操作。非幂等操作每次都改变状态。

那么更新 DOM 呢？

```
var hist = document.getElementById( "orderHistory" );

// 幂等的：
hist.innerHTML = order.historyText;

// 非幂等的：
var update = document.createTextNode( order.latestUpdate );
hist.appendChild( update );
```

这里的关键区别在于，幂等的更新替换了 DOM 元素的内容。DOM 元素的当前状态是独立的，因为它是无条件覆盖的。非幂等的操作将内容添加到元素中；隐式地，DOM 元素的当前状态是计算下一个状态的一部分。

我们将不会一直用幂等的方式去定义你的数据，但如果你能做到，这肯定会减少你的副作用在你最意想不到的时候突然出现的可能性。

纯粹的快乐

没有副作用的函数称为纯函数。在编程的意义上，纯函数是一种幂等函数，因为它不可能有任何副作用。思考一下：

```
function add(x,y) {
    return x + y;
}
```

所有输入（`x` 和 `y`）和输出（`return ..`）都是直接的，没有引用自由变量。调用 `add(3,4)` 多次和调用一次是没有区别的。`add(..)` 是纯粹的编程风格的幂等。

然而，并不是所有的纯函数都是数学概念上的幂等，因为它们返回的值不一定适合作为再次调用它们时的输入。思考一下：

```
function calculateAverage(list) {
    var sum = 0;
    for (let i = 0; i < list.length; i++) {
        sum += list[i];
    }
    return sum / list.length;
}

calculateAverage( [1,2,4,7,11,16,22] );            // 9
```

输出的 `9` 并不是一个数组，所以你不能在 `calculateAverage(calculateAverage(..))` 中将其传入。

正如我们前面所讨论的，一个纯函数可以引用自由变量，只要这些自由变量不是侧因。

例如：

```
const PI = 3.141592;

function circleArea(radius) {
    return PI * radius * radius;
}

function cylinderVolume(radius,height) {
    return height * circleArea( radius );
}
```

`circleArea(..)` 中引用了自由变量 `PI`，但是这是一个常量所以不是一个侧因。`cylinderVolume(..)` 引用了自由变量 `circleArea`，这也不是一个侧因，因为这个程序把它当作一个常量引用它的函数值。这两个函数都是纯的。

另一个例子，一个函数仍然可以是纯的，但引用的自由变量是闭包:

```
function unary(fn) {
    return function onlyOneArg(arg){
        return fn( arg );
    };
}
```

`unary(..)` 本身显然是纯函数 —— 它唯一的输入是 `fn`，并且它唯一的输出是返回的函数，但是闭合了自由变量 `fn` 的内部函数 `onlyOneArg(..)` 是不是纯的呢？

它仍然是纯的，因为 `fn` 永远不变。事实上，我们对这一事实有充分的自信，因为从词法上讲，这几行是唯一可能重新分配 `fn` 的代码。

注意： `fn` 是一个函数对象的引用，它默认是一个可变的值。在程序的其他地方可能为这个函数对象添加一个属性，这在技术上“改变”这个值（改变，而不是重新分配）。然而，因为我们除了调用 `fn`，不依赖 `fn` 以外的任何事情，并且不可能影响函数值的可调用性，因此 `fn` 在最后的结果中仍然是有效的不变的；它不可能是一个侧因。

表达一个函数的纯度的另一种常用方法是：给定相同的输入（一个或多个），它总是产生相同的输出。 如果你把 `3` 传给 `circleArea(..)` 它总是输出相同的结果（ `28.274328` ）。

如果一个函数每次在给予相同的输入时，可能产生不同的输出，那么它是不纯的。即使这样的函数总是返回相同的值，只要它产生间接输出副作用，并且程序状态每次被调用时都会被改变，那么这就是不纯的。

不纯的函数是不受欢迎的，因为它们使得所有的调用都变得更加难以理解。纯的函数的调用是完全可预测的。当有人阅读代码时，看到多个 `circleArea(3)` 调用，他们不需要花费额外的精力来计算每次的输出结果。

相对的纯粹

当我们讨论一个函数是纯的时，我们必须非常小心。JavaScript 的动态值特性使其很容易产生不明显的副作用。

思考一下：

```
function rememberNumbers(nums) {
    return function caller(fn){
        return fn( nums );
    };
}

var list = [1,2,3,4,5];

var simpleList = rememberNumbers( list );
```

`simpleList(..)` 看起来是一个纯函数，因为它只涉及内部的 `caller(..)` 函数，它仅仅是闭合了自由变量 `nums` 。然而，有很多方法证明 `simpleList(..)` 是不纯的。

首先，我们对纯度的断言是基于数组的值（通过 `list` 和 `nums` 引用）一直不改变：

```
function median(nums) {
    return (nums[0] + nums[nums.length - 1]) / 2;
}

simpleList( median );        // 3

// ..

list.push( 6 );

// ..

simpleList( median );        // 3.5
```

当我们改变数组时， `simpleList(..)` 的调用改变它的输出。所以， `simpleList(..)` 是纯的还是不纯的呢？这就取决于你的视角。对于给定的一组假设来说，它是纯函数。在任何没有 `list.push(6)` 的情况下是纯的。

我们可以通过改变 `rememberNumbers(..)` 的定义来修改这种不纯。一种方法是复制 `nums` 数组：

```
function rememberNumbers(nums) {
        // 复制一个数组
    nums = nums.slice();

    return function caller(fn){
        return fn( nums );
    };
}
```

但这可能会隐含一个更棘手的副作用：

```
var list = [1,2,3,4,5];

// 把 list[0] 作为一个有副作用的接收者
Object.defineProperty(
    list,
    0,
    {
        get: function(){
            console.log( "[0] was accessed!" );
            return 1;
        }
    }
);

var simpleList = rememberNumbers( list );

// [0] 已经被使用！
```

一个更粗鲁的选择是更改 `rememberNumbers(..)` 的参数。首先，不要接收数组，而是把数字作为单独的参数：

```
function rememberNumbers(...nums) {
    return function caller(fn){
        return fn( nums );
    };
}

var simpleList = rememberNumbers( ...list );

// [0] 已经被使用！
```

这两个 `...` 的作用是将列表复制到 `nums` 中，而不是通过引用来传递。

注意：控制台消息的副作用不是来自于 `rememberNumbers(..)`，而是 `...list` 的扩展中。因此，在这种情况下，`rememberNumbers(..)` 和 `simpleList(..)` 是纯的。

但是如果这种突变更难被发现呢？纯函数和不纯的函数的合成总是产生不纯的函数。如果我们将一个不纯的函数传递到另一个纯函数 `simpleList(..)` 中，那么这个函数就是不纯的：

```
// 是的，一个愚蠢的人为的例子 :)
function firstValue(nums) {
    return nums[0];
}

function lastValue(nums) {
    return firstValue( nums.reverse() );
}

simpleList( lastValue );    // 5

list;                        // [1,2,3,4,5] -- OK!

simpleList( lastValue );    // 1
```

注意： 不管 `reverse()` 看起来多安全（就像 JS 中的其他数组方法一样），它返回一个反向数组，实际上它对数组进行了修改，而不是创建一个新的数组。

我们需要对 `rememberNumbers(..)` 下一个更斩钉截铁的定义来防止 `fn(..)` 改变它的闭合的 `nums` 变量的引用。

```
function rememberNumbers(...nums) {
    return function caller(fn){
            // 提交一个副本！
        return fn( nums.slice() );
    };
}
```

所以 `simpleList(..)` 是可靠的纯函数吗！？不。 :(

我们只防范我们可以控制的副作用（通过引用改变）。我们传递的任何带有副作用的函数，都将会污染 `simpleList(..)` 的纯度:

```
simpleList( function impureIO(nums){
    console.log( nums.length );
} );
```

事实上，没有办法定义 `rememberNumbers(..)` 去产生一个完美纯粹的 `simpleList(..)` 函数。

纯度是和自信是有关的。但我们不得不承认，在很多情况下，我们所感受到的自信实际上是与我们程序的上下文和我们对程序了解有关的。在实践中（在 JavaScript 中），函数纯度的问题不是纯粹的纯粹性，而是关于其纯度的一系列信心。

越纯洁越好。制作纯函数时越努力，当您阅读使用它的代码时，你的自信就会越高，这将使代码的一部分更加可读。

有或者无

到目前为止，我们已经将函数纯度定义为一个没有副作用的函数，并且作为这样一个函数，给定相同的输入，总是产生相同的输出。这只是看待相同特征的两种不同方式。

但是，第三种看待函数纯性的方法，也许是广为接受的定义，即纯函数具有引用透明性。

引用透明性是指一个函数调用可以被它的输出值所代替，并且整个程序的行为不会改变。换句话说，不可能从程序的执行中分辨出函数调用是被执行的，还是它的返回值是在函数调用的位置上内联的。

从引用透明的角度来看，这两个程序都有完全相同的行为因为它们都是用纯粹的函数构建的：

```
function calculateAverage(list) {
    var sum = 0;
    for (let i = 0; i < list.length; i++) {
        sum += list[i];
    }
    return sum / list.length;
}

var nums = [1,2,4,7,11,16,22];

var avg = calculateAverage( nums );

console.log( "The average is:", avg );        // The average is: 9
```

```
function calculateAverage(list) {
    var sum = 0;
    for (let i = 0; i < list.length; i++) {
        sum += list[i];
    }
    return sum / list.length;
}

var nums = [1,2,4,7,11,16,22];

var avg = 9;

console.log( "The average is:", avg );        // The average is: 9
```

这两个片段之间的唯一区别在于，在后者中，我们跳过了调用 `calculateAverage(nums)` 并内联。因为程序的其他部分的行为是相同的，`calculateAverage(..)` 是引用透明的，因此是一个纯粹的函数。

思考上的透明

一个引用透明的纯函数可能会被它的输出替代，这并不意味着它应该被替换。远非如此。

我们用在程序中使用函数而不是使用预先计算好的常量的原因不仅仅是应对变化的数据，也是和可读性和适当的抽象等有关。调用函数去计算一列数字的平均值让这部分程序比只是使用确定的值更具有可读性。它向读者讲述了 `avg` 从何而来，它意味着什么，等等。

我们真正建议使用引用透明是当你阅读程序，一旦你已经在内心计算出纯函数调用输出的是什么的时候，当你看到它的代码的时候不需要再去思考确切的函数调用是做什么，特别是如果它出现很多次。

这个结果有一点像你在心里面定义一个 `const`，当你阅读的时候，你可以直接跳过并且不需要花更多的精力去计算。

我们希望纯函数的这种特性的重要性是显而易见的。我们正在努力使我们的程序更容易读懂。我们能做的一种方法是给读者较少的工作，通过提供帮助来跳过不必要的东西，这样他们就可以把注意力集中在重要的事情上。

读者不需要重新计算一些不会改变（也不需要改变）的结果。如果用引用透明定义一个纯函数，读者就不必这样做了。

不够透明？

那么如果一个有副作用的函数，并且这个副作用在程序的其他地方没有被观察到或者依赖会怎么样？这个功能还具有引用透明性吗？

这里有一个例子：

```
function calculateAverage(list) {
    sum = 0;
    for (let i = 0; i < list.length; i++) {
        sum += list[i];
    }
    return sum / list.length;
}

var sum, nums = [1,2,4,7,11,16,22];

var avg = calculateAverage( nums );
```

你发现了吗?

`sum` 是一个 `calculateAverage(..)` 使用的外部自由变量。但是，每次我们使用相同的列表调用 `calculateAverage(..)`，我们将得到 `9` 作为输出。并且这个程序无法和使用参数 `9` 调用 `calculateAverage（nums）` 在行为上区分开来。程序的其他部分和 `sum` 变量有关，所以这是一个不可观察的副作用。 这是一个像这棵树一样不能观察到的副作用吗？

> 假如一棵树在森林里倒下而没有人在附近听见，它有没有发出声音？

通过引用透明的狭义的定义，我想你一定会说 `calculateAverage(..)` 仍然是一个纯函数。但是，因为在我们的学习中不仅仅是学习学术，而且与实用主义相平衡，我认为这个结论需要更多的观点。让我们探索一下。

性能影响

你经常会发现这些不易观察的副作用被用于性能优化的操作。例如：

```
var cache = [];

function specialNumber(n) {
        // 如果我们已经计算过这个特殊的数，
    // 跳过这个操作，然后从缓存中返回
    if (cache[n] !== undefined) {
        return cache[n];
    }

    var x = 1, y = 1;

    for (let i = 1; i <= n; i++) {
        x += i % 2;
        y += i % 3;
    }

    cache[n] = (x * y) / (n + 1);

    return cache[n];
}

specialNumber( 6 );                // 4
specialNumber( 42 );            // 22
specialNumber( 1E6 );            // 500001
specialNumber( 987654321 );        // 493827162
```

这个愚蠢的 `specialNumber(..)` 算法是确定性的，并且，纯函数从定义来说，它总是为相同的输入提供相同的输出。从引用透明的角度来看 —— 用 `22` 替换对 `specialNumber(42)` 的任何调用，程序的最终结果是相同的。

但是，这个函数必须做一些工作来计算一些较大的数字，特别是输入像 `987654321` 这样的数字。如果我们需要在我们的程序中多次获得特定的特殊号码，那么结果的缓存意味着后续的调用效率会更高。

注意： 思考一个有趣的事情：CPU 在执行任何给定操作时产生的热量，即使是最纯粹的函数 / 程序，也是不可避免的副作用吗？那么 CPU 的时间延迟，因为它花时间在一个纯操作上，然后再执行另一个操作，是否也算作副作用？

不要这么快地做出假设，你仅仅运行 `specialNumber（987654321）` 计算一次，并手动将该结果粘贴到一些变量 / 常量中。程序通常是高度模块化的并且全局可访问的作用域并不是通常你想要在这些独立部分之间分享状态的方式。让 `specialNumber（..）` 使用自己的缓存（即使它恰好是使用一个全局变量来实现这一点）是对状态共享更好的抽象。

关键是，如果 `specialNumber(..)` 只是程序访问和更新 `cache` 副作用的唯一部分，那么引用透明的观点显然可以适用，这可以被看作是可以接受的实际的“欺骗”的纯函数思想。

但是真的应该这样吗？

典型的，这种性能优化方面的副作用是通过隐藏缓存结果产生的，因此它们不能被程序的任何其他部分所观察到。这个过程被称为记忆化。我一直称这个词是 “记忆化”，我不知道这个想法是从哪里来的，但它确实有助于我更好地理解这个概念。

思考一下：

```
var specialNumber = (function memoization(){
    var cache = [];

    return function specialNumber(n){
            // 如果我们已经计算过这个特殊的数，
            // 跳过这个操作，然后从缓存中返回
        if (cache[n] !== undefined) {
            return cache[n];
        }

        var x = 1, y = 1;

        for (let i = 1; i <= n; i++) {
            x += i % 2;
            y += i % 3;
        }

        cache[n] = (x * y) / (n + 1);

        return cache[n];
    };
})();
```

我们已经遏制 `memoization()` 内部 `specialNumber(..)` IIFE 范围内的 `cache` 的副作用，所以现在我们确定程序任何的部分都不能观察到它们，而不仅仅是不观察它们。

最后一句话似乎是一个的微妙观点，但实际上我认为这可能是整章中最重要的一点。 再读一遍。

回到这个哲学理论：

> 假如一棵树在森林里倒下而没有人在附近听见，它有没有发出声音？

通过这个暗喻，我所得到的是：无论是否产生声音，如果我们从不创造一个当树落下时周围没有人的情景会更好一些。当树落下时，我们总是会听到声音。

减少副作用的目的并不是他们在程序中不能被观察到，而是设计一个程序，让副作用尽可能的少，因为这使代码更容易理解。一个没有观察到的发生的副作用的程序在这个目标上并不像一个不能观察它们的程序那么有效。

如果副作用可能发生，作者和读者必须尽量应对它们。使它们不发生，作者和读者都要对任何可能或不可能发生的事情更有自信。

纯化

如果你有不纯的函数，且你无法将其重构为纯函数，此时你能做些什么？

您需要确定该函数有什么样的副作用。副作用来自不同的地方，可能是由于词法自由变量、引用变化，甚至是 `this` 的绑定。我们将研究解决这些情况的方法。

封闭的影响

如果副作用的本质是使用词法自由变量，并且您可以选择修改周围的代码，那么您可以使用作用域来封装它们。

回忆一下：

```
var users = {};

function fetchUserData(userId) {
    ajax( "http://some.api/user/" + userId, function onUserData(userData){
        users[userId] = userData;
    } );
}
```

纯化此代码的一个方法是在变量和不纯的函数周围创建一个容器。本质上，容器必须接收所有的输入。

```
function safer_fetchUserData(userId,users) {
        // 简单的、原生的 ES6 + 浅拷贝，也可以
    // 用不同的库或框架
    users = Object.assign( {}, users );

    fetchUserData( userId );

        // 返回拷贝过的状态
    return users;


    // ***********************

        // 原始的没被改变的纯函数：
    function fetchUserData(userId) {
        ajax( "http://some.api/user/" + userId, function onUserData(userData){
            users[userId] = userData;
        } );
    }
}
```

`userId` 和 `users` 都是原始的的 `fetchUserData` 的输入，`users` 也是输出。`safer_fetchUserData(..)` 取出他们的输入，并返回 `users`。为了确保在 `users` 被改变时我们不会在外部创建副作用，我们制作一个 `users` 本地副本。

这种技术的有效性有限，主要是因为如果你不能将函数本身改为纯的，你也几乎不可能修改其周围的代码。然而，如果可能，探索它是有帮助的，因为它是所有修复方法中最简单的。

无论这是否是重构纯函数的一个实际方法，最重要的是函数的纯度仅仅需要深入到皮肤。也就是说，函数的纯度是从外部判断的， 不管内部是什么。只要一个函数的使用表现为纯的，它就是纯的。在纯函数的内部，由于各种原因，包括最常见的性能方面，可以适度的使用不纯的技术。正如他们所说的“世界是一只驮着一只一直驮下去的乌龟群”。

不过要小心。程序的任何部分都是不纯的，即使它仅仅是用纯函数包裹的，也是代码错误和困惑读者的潜在的根源。总体目标是尽可能减少副作用，而不仅仅是隐藏它们。

覆盖效果

很多时候，你无法在容器函数的内部为了封装词法自由变量来修改代码。例如，不纯的函数可能位于一个你无法控制的第三方库文件中，其中包括：

```
var nums = [];
var smallCount = 0;
var largeCount = 0;

function generateMoreRandoms(count) {
    for (let i = 0; i < count; i++) {
        let num = Math.random();

        if (num >= 0.5) {
            largeCount++;
        }
        else {
            smallCount++;
        }

        nums.push( num );
    }
}
```

蛮力的策略是，在我们程序的其余部分使用此通用程序时隔离副作用的方法时创建一个接口函数，执行以下步骤：

1. 捕获受影响的当前状态
2. 设置初始输入状态
3. 运行不纯的函数
4. 捕获副作用状态
5. 恢复原来的状态
6. 返回捕获的副作用状态

```
function safer_generateMoreRandoms(count,initial) {
        // (1) 保存原始状态
    var orig = {
        nums,
        smallCount,
        largeCount
    };

        // (2) 设置初始副作用状态
    nums = initial.nums.slice();
    smallCount = initial.smallCount;
    largeCount = initial.largeCount;

        // (3) 当心杂质！
    generateMoreRandoms( count );

        // (4) 捕获副作用状态
    var sides = {
        nums,
        smallCount,
        largeCount
    };

        // (5) 重新存储原始状态
    nums = orig.nums;
    smallCount = orig.smallCount;
    largeCount = orig.largeCount;

        // (6) 作为输出直接暴露副作用状态
    return sides;
}
```

并且使用 `safer_generateMoreRandoms(..)`：

```
var initialStates = {
    nums: [0.3, 0.4, 0.5],
    smallCount: 2,
    largeCount: 1
};

safer_generateMoreRandoms( 5, initialStates );
// { nums: [0.3,0.4,0.5,0.8510024448959794,0.04206799238...

nums;                 // []
smallCount;             // 0
largeCount;             // 0
```

这需要大量的手动操作来避免一些副作用，如果我们一开始就没有它们，那就容易多了。但如果我们别无选择，那么这种额外的努力是值得的，以避免我们的项目出现意外。

注意： 这种技术只有在处理同步代码时才有用。异步代码不能可靠地使用这种方法被管理，因为如果程序的其他部分在期间也在访问 / 修改状态变量，它就无法防止意外。

回避影响

当要处理的副作用的本质是直接输入值（对象、数组等）的突变时，我们可以再次创建一个接口函数来替代原始的不纯的函数去交互。

考虑一下：

```
function handleInactiveUsers(userList,dateCutoff) {
    for (let i = 0; i < userList.length; i++) {
        if (userList[i].lastLogin == null) {
                // 将 user 从 list 中删除
            userList.splice( i, 1 );
            i--;
        }
        else if (userList[i].lastLogin < dateCutoff) {
            userList[i].inactive = true;
        }
    }
}
```

`userList` 数组本身，加上其中的对象，都发生了改变。防御这些副作用的一种策略是先做一个深拷贝（不是浅拷贝）：

```
function safer_handleInactiveUsers(userList,dateCutoff) {
        // 拷贝列表和其中 `user` 的对象
    let copiedUserList = userList.map( function mapper(user){
            // 拷贝 user 对象
        return Object.assign( {}, user );
    } );

        // 使用拷贝过的对象调用最初的函数
    handleInactiveUsers( copiedUserList, dateCutoff );

    // 将突变的 list 作为直接的输出暴露出来
    return copiedUserList;
}
```

这个技术的成功将取决于你所做的复制的深度。使用 `userList.slice()` 在这里不起作用，因为这只会创建一个 `userList` 数组本身的浅拷贝。数组的每个元素都是一个需要复制的对象，所以我们需要格外小心。当然，如果这些对象在它们之内有对象（可能会这样），则复制需要更加完善。

再看一下 `this`

另一个参数变化的副作用是和 `this` 有关的，我们应该意识到 `this` 是函数隐式的输入。查看第 2 章中的“什么是This”获取更多的信息，为什么 `this` 关键字对函数式编程者是不确定的。

思考一下：

```
var ids = {
    prefix: "_",
    generate() {
        return this.prefix + Math.random();
    }
};
```

我们的策略类似于上一节的讨论：创建一个接口函数，强制 `generate()` 函数使用可预测的 `this` 上下文：

```
function safer_generate(context) {
    return ids.generate.call( context );
}

// *********************

safer_generate( { prefix: "foo" } );
// "foo0.8988802158307285"
```

这些策略绝对不是愚蠢的，对副作用的最安全的保护是不要产生它们。但是，如果您想提高程序的可读性和你对程序的自信，无论在什么情况下尽可能减少副作用 / 效果是巨大的进步。

本质上，我们并没有真正消除副作用，而是克制和限制它们，以便我们的代码更加的可验证和可靠。如果我们后来遇到程序错误，我们就知道代码仍然产生副作用的部分最有可能是罪魁祸首。

总结

副作用对代码的可读性和质量都有害，因为它们使您的代码难以理解。副作用也是程序中最常见的错误原因之一，因为很难应对他们。幂等是通过本质上创建仅有一次的操作来限制副作用的一种策略。

避免副作用的最优方法是使用纯函数。纯函数给定相同输入时总返回相同输出，并且没有副作用。引用透明更近一步的状态是 —— 更多的是一种脑力运动而不是文字行为 —— 纯函数的调用是可以用它的输出来代替，并且程序的行为不会被改变。

将一个不纯的函数重构为纯函数是首选。但是，如果无法重构，尝试封装副作用，或者创建一个纯粹的接口来解决问题。

没有程序可以完全没有副作用。但是在实际情况中的很多地方更喜欢纯函数。尽可能地收集纯函数的副作用，这样当错误发生时更加容易识别和审查出最像罪魁祸首的错误。

JavaScript 轻量级函数式编程

第 6 章：值的不可变性

在第 5 章中，我们探讨了减少副作用的重要性：副作用是引起程序意外状态改变的原因，同时也可能会带来意想不到的惊喜（bugs）。这样的暗雷在程序中出现的越少，开发者对程序的信心无疑就会越强，同时代码的可读性也会越高。本章的主题，将继续朝减少程序副作用的方向努力。

如果编程风格幂等性是指定义一个数据变更操作以便只影响一次程序状态，那么现在我们将注意力转向将这个影响次数从 1 降为 0。

现在我们开始探索值的不可变性，即只在我们的程序中使用不可被改变的数据。

原始值的不可变性

原始数据类型（`number`、`string`、`boolean`、`null` 和 `undefined`）本身就是不可变的；无论如何你都没办法改变它们。

```
// 无效，且毫无意义
2 = 2.5;
```

然而 JS 确实有一个特性，使得看起来允许我们改变原始数据类型的值, 即“boxing”特性。当你访问原始类型数据时 —— 特别是 `number`、`string` 和 `boolean` —— 在这种情况下，JS 会自动的把它们包裹（或者说“包装”）成这个值对应的对象（分别是 `Number`、`String` 以及 `Boolean`）。

思考下面的代码：

```
var x = 2;

x.length = 4;

x;                  // 2
x.length;          // undefined
```

数值本身并没有可用的 `length` 属性，因此 `x.length = 4` 这个赋值操作正试图添加一个新的属性，不过它静默地失败了（也可以说是这个操作被忽略了或被抛弃了，这取决于你怎么看）；变量 `x` 继续承载那个简单的原始类型数据 —— 数值 `2`。

但是 JS 允许 `x.length = 4` 这条语句正常执行的事实着实令人困惑。如果这种现象真的无缘无故出现，那么代码的阅读者无疑会摸不着头脑。好消息是，如果你使用了严格模式（`"use strict";`），那么这条语句就会抛出异常了。

那么如果尝试改变那些明确被包装成对象的值呢？

```
var x = new Number( 2 );

// 没问题
x.length = 4;
```

这段代码中的 `x` 保存了一个对象的引用，因此可以正常地添加或修改自定义属性。

像 `number` 这样的原始数型，值的不可变性看起来相当明显，但字符串呢？JS 开发者有个共同的误解 —— 字符串和数组很像，所以应该是可变的。JS 使用 `[]` 访问字符串成员的语法甚至还暗示字符串真的就像数组。不过，字符串的确是不可变的：

```
var s = "hello";

s[1];                // "e"

s[1] = "E";
s.length = 10;

s;                    // "hello"
```

尽管可以使用 `s[1]` 来像访问数组元素一样访问字符串成员，JS 字符串也并不是真的数组。`s[1] = "E"` 和 `s.length = 10` 这两个赋值操作都是失败的，就像刚刚的 `x.length = 4` 一样。在严格模式下，这些赋值都会抛出异常，因为 `1` 和 `length` 这两个属性在原始数据类型字符串中都是只读的。

有趣的是，即便是包装后的 `String` 对象，其值也会（在大部分情况下）表现的和非包装字符串一样 —— 在严格模式下如果改变已存在的属性，就会抛出异常：

```
"use strict";

var s = new String( "hello" );

s[1] = "E";             // error
s.length = 10;        // error

s[42] = "?";        // OK

s;                    // "hello"
```

从值到值

我们将在本节详细展开从值到值这个概念。但在开始之前应该心中有数：值的不可变性并不是说我们不能在程序编写时不改变某个值。如果一个程序的内部状态从始至终都保持不变，那么这个程序肯定相当无趣！它同样不是指变量不能承载不同的值。这些都是对值的不可变这个概念的误解。

值的不可变性是指当需要改变程序中的状态时，我们不能改变已存在的数据，而是必须创建和跟踪一个新的数据。

例如：

```
function addValue(arr) {
    var newArr = [ ...arr, 4 ];
    return newArr;
}

addValue( [1,2,3] );    // [1,2,3,4]
```

注意我们没有改变数组 `arr` 的引用，而是创建了一个新的数组（ `newArr` ），这个新数组包含数组 `arr` 中已存在的值，并且新增了一个新值 `4` 。

使用我们在第 5 章讨论的副作用的相关概念来分析 `addValue(..)` 。它是纯的吗？它是否具有引用透明性？给定相同的数组作为输入，它会永远返回相同的输出吗？它无副作用吗？答案是肯定的。

设想这个数组 `[1, 2, 3]` , 它是由先前的操作产生，并被我们保存在一个变量中，它代表着程序当前的状态。我们想要计算出程序的下一个状态，因此调用了 `addValue(..)` 。但是我们希望下一个状态计算的行为是直接的和明确的，所以 `addValue(..)` 操作简单的接收一个直接输入，返回一个直接输出，并通过不改变 `arr` 引用的原始数组来避免副作用。

这就意味着我们既可以计算出新状态 `[1, 2, 3, 4]` ，也可以掌控程序的状态变换。程序不会出现过早的过渡到这个状态或完全转变到另一个状态（如 `[1, 2, 3, 5]` ）这样的意外情况。通过规范我们的值并把它视为不可变的，我们大幅减少了程序错误，使我们的程序更易于阅读和推导，最终使程序更加可信赖。

`arr` 所引用的数组是可变的，只是我们选择不去改变他，我们实践了值不可变的这一精神。

同样的，可以将“以拷贝代替改变”这样的策略应用于对象，思考下面的代码：

```
function updateLastLogin(user) {
    var newUserRecord = Object.assign( {}, user );
    newUserRecord.lastLogin = Date.now();
    return newUserRecord;
}

var user = {
    // ..
};

user = updateLastLogin( user );
```

消除本地影响

下面的代码能够体现不可变性的重要性：

```
var arr = [1,2,3];

foo( arr );

console.log( arr[0] );
```

从表面上讲，你可能认为 `arr[0]` 的值仍然为 `1`。但事实是否如此不得而知，因为 `foo(..)` 可能会改变你传入其中的 `arr` 所引用的数组。

在之前的章节中，我们已经见到过用下面这种带有欺骗性质的方法来避免意外：

```
var arr = [1,2,3];

foo( arr.slice() );            // 哈！一个数组副本！

console.log( arr[0] );        // 1
```

当然，使得这个断言成立的前提是 `foo` 函数不会忽略我们传入的参数而直接通过相同的 `arr` 这个自由变量词法引用来访问源数组。

对于防止数据变化负面影响，稍后我们会讨论另一种策略。

重新赋值

在进入下一个段落之前先思考一个问题 —— 你如何描述“常量”？

...

你可能会脱口而出“一个不能改变的值就是常量”，“一个不能被改变的变量”等等。这些回答都只能说接近正确答案，但却并不是正确答案。对于常量，我们可以给出一个简洁的定义：一个无法进行重新赋值（reassignment）的变量。

我们刚刚在“常量”概念上的吹毛求疵其实是很有必要的，因为它澄清了常量与值无关的事实。无论常量承载何值，该变量都不能使用其他的值被进行重新赋值。但它与值的本质无关。

思考下面的代码：

```
var x = 2;
```

我们刚刚讨论过，数据 `2` 是一个不可变的原始值。如果将上面的代码改为：

```
const x = 2;
```

`const` 关键字的出现，作为“常量声明”被大家熟知，事实上根本没有改变 `2` 的本质，因为它本身就已经不可改变了。

下面这行代码会抛出错误，这无可厚非：

```
// 尝试改变 x，祝我好运！
x = 3;          // 抛出错误！
```

但再次重申，我们并不是要改变这个数据，而是要对变量 `x` 进行重新赋值。数据被卷进来纯属偶然。

为了证明 `const` 和值的本质无关，思考下面的代码：

```
const x = [ 2 ];
```

这个数组是一个常量吗？并不是。 `x` 是一个常量，因为它无法被重新赋值。但下面的操作是完全可行的：

```
x[0] = 3;
```

为何？因为尽管 `x` 是一个常量，数组却是可变的。

关于 `const` 关键字和“常量”只涉及赋值而不涉及数据语义的特性是个又臭又长的故事。几乎所有语言的高级开发者都踩 `const` 地雷。事实上，Java 最终不赞成使用 `const` 并引入了一个全新的关键词 `final` 来区分“常量”这个语义。

抛开混乱之后开始思考，如果 `const` 并不能创建一个不可变的值，那么它对于函数式编程者来说又还有什么重要的呢？

意图

`const` 关键字可以用来告知阅读你代码的读者该变量不会被重新赋值。作为一个表达意图的标识，`const` 被加入 JavaScript 不仅常常受到称赞，也普遍提高了代码可读性。

在我看来，这是夸大其词，这些说法并没有太大的实际意义。我只看到了使用这种方法来表明意图的微薄好处。如果使用这种方法来声明值的不可变性，与已使用几十年的传统方式相比，`const` 简直太弱了。

为了证明我的说法，让我们来做一个实践。`const` 创建了一个在块级作用域内的变量，这意味着该变量只能在其所在的代码块中被访问：

```
// 大量代码

{
    const x = 2;

    // 少数几行代码
}

// 大量代码
```

通常来说，代码块的最佳实践是用于仅包裹少数几行代码的场景。如果你有一个包含了超过 10 行的代码块，那么大多数开发者会建议你重构这一段代码。因此 `const x = 2` 只作用于下面的9行代码。

程序的其他部分不会影响 `x` 的赋值。

我要说的是：上述程序的可读性与下面这样基本相同：

```
// 大量代码

{
    let x = 2;

    // 少数几行代码
}

// 大量代码
```

其实只要查看一下在 `let x = 2`；之后的几行代码，就可以判断出 x 这个变量是否被重新赋值过了。对我来说，“实际上不进行重新赋值”相对“使用容易迷惑人的 `const` 关键字告诉读者‘不要重新赋值’”是一个更明确的信号。

此外，让我们思考一下，乍看这段代码起来可能给读者传达什么：

```
const magicNums = [1,2,3,4];

// ..
```

读者可能会（错误地）认为，这里使用 `const` 的用意是你永远不会修改这个数组 —— 这样的推断对我来说合情合理。想象一下，如果你的确允许 `magicNums` 这个变量所引用的数组被修改，那么这个 `const` 关键词就极具混淆性了 —— 的很确容易发生意外，不是吗？

更糟糕的是，如果你在某处故意修改了 `magicNums`，但对读者而言不够明显呢？读者会在后面的代码里（再次错误地）认为 `magicNums` 的值仍然是 `[1, 2, 3, 4]`。因为他们猜测你之前使用 `const` 的目的就是“这个变量不会改变”。

我认为你应该使用 `var` 或 `let` 来声明那些你会去改变的变量，它们确实相比 `const` 来说是一个更明确的信号。

`const` 所带来的问题还没讲完。还记得我们在本章开头所说的吗？值的不可变性是指当需要改变某个数据时，我们不应该直接改变它，而是应该使用一个全新的数据。那么当新数组创建出来后，你会怎么处理它？如果你使用 `const` 声明变量来保存引用吗，这个变量的确没法被重新赋值了，那么……然后呢？

从这方面来讲，我认为 `const` 反而增加了函数式编程的困难度。我的结论是：`const` 并不是那么有用。它不仅造成了不必要的混乱，也以一种很不方便的形式限制了我们。我只用 `const` 来声明简单的常量，例如：

```
const PI = 3.141592;
```

`3.141592` 这个值本身就已经是不可变的，并且我也清楚地表示说“`PI` 标识符将始终被用于代表这个字面量的占位符”。对我来说，这才是 `const` 所擅长的。坦白讲，我在编码时并不会使用很多这样的声明。

我写过很多，也阅读过很多 JavaScript 代码，我认为由于重新赋值导致大量的 bug 这只是个想象中的问题，实际并不存在。

我们应该担心的，并不是变量是否被重新赋值，而是值是否会发生改变。为什么？因为值是可被携带的，但词法赋值并不是。你可以向函数中传入一个数组，这个数组可能会在你没意识到的情况下被改变。但是你的其他代码在预期之外重新给变量赋值，这是不可能发生的。

冻结

这是一种简单廉价的（勉强）将像对象、数组、函数这样的可变的数据转为“不可变数据”的方式：

```
var x = Object.freeze( [2] );
```

`Object.freeze(..)` 方法遍历对象或数组的每个属性和索引，将它们设置为只读以使之不会被重新赋值，事实上这和使用 `const` 声明属性相差无几。`Object.freeze(..)` 也会将属性标记为“不可配置（non-reconfigurable）”，并且使对象或数组本身不可扩展（即不会被添加新属性）。实际上，而就可以将对象的顶层设为不可变。

注意，仅仅是顶层不可变！

```
var x = Object.freeze( [ 2, 3, [4, 5] ] );

// 不允许改变：
x[0] = 42;

// oops，仍然允许改变：
x[2][0] = 42;
```

`Object.freeze(..)` 提供浅层的、初级的不可变性约束。如果你希望更深层的不可变约束，那么你就得手动遍历整个对象或数组结构来为所有后代成员应用 `Object.freeze(..)` 。

与 `const` 相反，`Object.freeze(..)` 并不会误导你，让你得到一个“你以为”不可变的值，而是真真确确给了你一个不可变的值。

回顾刚刚的例子：

```
var arr = Object.freeze( [1,2,3] );

foo( arr );

console.log( arr[0] );          // 1
```

可以非常确定 `arr[0]` 就是 `1` 。

这是非常重要的，因为这可以使我们更容易的理解代码，当我们将值传递到我们看不到或者不能控制的地方，我们依然能够相信这个值不会改变。

性能

每当我们开始创建一个新值（数组、对象等）取代修改已经存在的值时，很明显迎面而来的问题就是：这对性能有什么影响？

如果每次想要往数组中添加内容时，我们都必须创建一个全新的数组，这不仅占用 CPU 时间并且消耗额外的内存。不再存在任何引用的旧数据将会被垃圾回收机制回收；更多的 CPU 资源消耗。

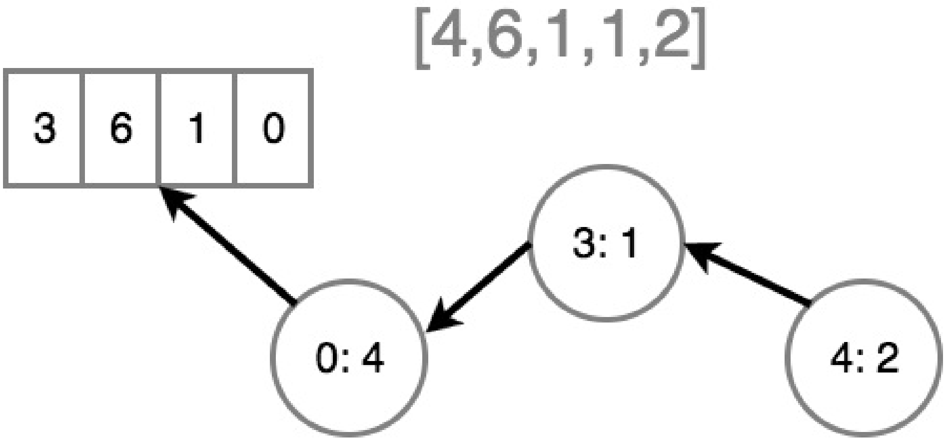
这样的取舍能接受吗？视情况而定。对代码性能的优化和讨论都应该有个上下文。

如果在你的程序中，只会发生一次或几次单一的状态变化，那么扔掉一个旧对象或旧数组完全没必要担心。性能损失会非常非常小 —— 顶多只有几微秒 —— 对你的应用程序影响甚小。追踪和修复由于数据改变引起的 bug 可能会花费你几分钟甚至几小时的时间，这么看来那几微秒简直没有可比性。

但是，如果频繁的进行这样的操作，或者这样的操作出现在应用程序的核心逻辑中，那么性能问题 —— 即性能和内存 —— 就有必要仔细考虑一下了。

以数组这样一个特定的数据结构来说，我们想要在每次操作这个数组时使每个更改都隐式地进行，就像结果是一个新数组一样，但除了每次都真的创建一个数组之外，还有什么其他办法来完成这个任务呢？像数组这样的数据结构，我们期望除了能够保存其最原始的数据，然后能追踪其每次改变并根据之前的版本创建一个分支。

在内部，它可能就像一个对象引用的链表树，树中的每个节点都表示原始值的改变。从概念上来说，这和 **git** 的版本控制原理类似。



想象一下使用这个假设的、专门处理数组的数据结构：

```
var state = specialArray( 1, 2, 3, 4 );

var newState = state.set( 42, "meaning of life" );

state === newState;                     // false

state.get( 2 );                         // 3
state.get( 42 );                     // undefined

newState.get( 2 );                     // 3
newState.get( 42 );                     // "meaning of life"

newState.slice( 1, 3 );                 // [2,3]
```

`specialArray(..)` 这个数据结构会在内部追踪每个数据更新操作（例如 `set(..)` ），类似 *diff*，因此不必要为原始的那些值（ `1` 、 `2` 、 `3` 和 `4` ）重新分配内存，而是简单的将 `"meaning of life"` 这个值加入列表。重要的是， `state` 和 `newState` 分别指向两个“不同版

本"的数组，因此值的不变性这个语义得以保留。

发明你自己的性能优化数据结构是个有趣的挑战。但从实用性来讲，找一个现成的库会是个更好的选择。**Immutable.js**（http://facebook.github.io/immutable-js） 是一个很棒的选择，它提供多种数据结构，包括 `List` （类似数组）和 `Map` （类似普通对象）。

思考下面的 `specialArray` 示例，这次使用 `Immutable.List` ：

```
var state = Immutable.List.of( 1, 2, 3, 4 );

var newState = state.set( 42, "meaning of life" );

state === newState;                     // false

state.get( 2 );                         // 3
state.get( 42 );                     // undefined

newState.get( 2 );                     // 3
newState.get( 42 );                     // "meaning of life"

newState.toArray().slice( 1, 3 );    // [2,3]
```

像 Immutable.js 这样强大的库一般会采用非常成熟的性能优化。如果不使用库而是手动去处理那些细枝末节，开发的难度会相当大。

当改变值这样的场景出现的较少且不用太关心性能时，我推荐使用更轻量级的解决方案，例如我们之前提到过的内置的 `Object.freeze(..)` 。

以不可变的眼光看待数据

如果我们从函数中接收了一个数据，但不确定这个数据是可变的还是不可变的，此时该怎么办？去修改它试试看吗？不要这样做。 就像在本章最开始的时候所讨论的，不论实际上接收到的值是否可变，我们都应以它们是不可变的来对待，以此来避免副作用并使函数保持纯度。

回顾一下之前的例子：

```
function updateLastLogin(user) {
    var newUserRecord = Object.assign( {}, user );
    newUserRecord.lastLogin = Date.now();
    return newUserRecord;
}
```

该实现将 `user` 看做一个不应该被改变的数据来对待； `user` 是否真的不可变完全不会影响这段代码的阅读。对比一下下面的实现：

```
function updateLastLogin(user) {
    user.lastLogin = Date.now();
    return user;
}
```

这个版本更容易实现，性能也会更好一些。但这不仅让 `updateLastLogin(..)` 变得不纯，这种方式改变的值使阅读该代码，以及使用它的地方变得更加复杂。

应当总是将 **user** 看做不可变的值，这样我们就没必要知道数据从哪里来，也没必要担心数据改变会引发潜在问题。

JavaScript 中内置的数组方法就是一些很好的例子，例如 `concat(..)` 和 `slice(..)` 等：

```
var arr = [1,2,3,4,5];

var arr2 = arr.concat( 6 );

arr;                    // [1,2,3,4,5]
arr2;                    // [1,2,3,4,5,6]

var arr3 = arr2.slice( 1 );

arr2;                    // [1,2,3,4,5,6]
arr3;                    // [2,3,4,5,6]
```

其他一些将参数看做不可变数据且返回新数组的原型方法还有：`map(..)` 和 `filter(..)` 等。`reduce(..)` / `reduceRight(..)` 方法也会尽量避免改变参数，尽管它们并不默认返回新数组。

不幸的是，由于历史问题，也有一部分不纯的数组原型方法：`splice(..)` 、`pop(..)` 、`push(..)` 、`shift(..)` 、`unshift(..)` 、`reverse(..)` 以及 `fill(..)` 。

有些人建议禁止使用这些不纯的方法，但我不这么认为。因为一些性能面的原因，某些场景下你仍然可能会用到它们。不过你也应当注意，如果一个数组没有被本地化在当前函数的作用域内，那么不应当使用这些方法，避免它们所产生的副作用影响到代码的其他部分。

不论一个数据是否是可变的，永远将他们看做不可变。遵守这样的约定，你程序的可读性和可信赖度将会大大提升。

总结

值的不可变性并不是不改变值。它是指在程序状态改变时，不直接修改当前数据，而是创建并追踪一个新数据。这使得我们在读代码时更有信心，因为我们限制了状态改变的场景，状态不会在意料之外或不易观察的地方发生改变。

由于其自身的信号和意图，`const` 关键字声明的常量通常被误认为是强制规定数据不可被改变。事实上，`const` 和值的不可变性声明无关，而且使用它所带来的困惑似乎比它解决的问题还要大。另一种思路，内置的 `Object.freeze(..)` 方法提供了顶层值的不可变性设定。大多数情况下，使用它就足够了。

对于程序中性能敏感的部分，或者变化频繁发生的地方，处于对计算和存储空间的考量，每次都创建新的数据或对象（特别是在数组或对象包含很多数据时）是非常不可取的。遇到这种情况，通过类似 **Immutable.js** 的库使用不可变数据结构或许是个很棒的主意。

值不变在代码可读性上的意义，不在于不改变数据，而在于以不可变的眼光看待数据这样的约束。

JavaScript 轻量级函数式编程

第 7 章: 闭包 vs 对象

数年前，Anton van Straaten 创造了一个非常有名且被常常引用的 禅理 来举例和证实一个闭包和对象之间重要的关系。

> 德高望重的大师 Qc Na 曾经和他的学生 Anton 一起散步。Anton 希望引导大师到一个讨论里，说到：大师，我曾听说对象是一个非常好的东西，是这样么？Qc Na 同情地看着他的学生回答到，“愚笨的弟子，对象只不过是可怜人的闭包”
>
> 被批评后，Anton 离开他的导师并回到了自己的住处，致力于学习闭包。他认真的阅读整个“匿名函数：终极……”系列论文和它的姐妹篇，并且实践了一个基于闭包系统的小的 Scheme 解析器。他学了很多，盼望展现给他导师他的进步。
>
> 当他下一次与 Qc Na 一同散步时，Anton 试着提醒他的导师，说到 “导师，我已经勤奋地学习了这件事，我现在明白了对象真的是可怜人的闭包。” ，Qc Na 用棍子戳了戳 Anton 回应到，“你什么时候才能学会，闭包才是可怜人的对象”。在那一刻， Anton 明白了什么。
>
> Anton van Straaten 6/4/2003
>
> http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html

原帖尽管简短，却有更多关于起源和动机的内容，我强烈推荐为了理解本章去阅读原帖来调整你的观念。

我观察到很多人读完这个会对其中的聪明智慧傻笑，却继续不改变他们的想法。但是，这个禅理（来自 Bhuddist Zen 观点）促使读者进入其中对立真相的辩驳中。所以，返回并且再读一遍。

到底是哪个？是闭包是可怜的对象，还是对象是可怜的闭包？或都不是？或都是？或者这只是为了说明闭包和对象在某些方面是相同的方式？

还有它们中哪个与函数式编程相关？拉一把椅子过来并且仔细考虑一会儿。如果你愿意，这一章将是一个精彩的迂回之路，一个远足。

达成共识

先确定一点，当我们谈及闭包和对象我们都达成了共识。我们显然是在 JavaScript 如何处理这两种机制的上下文中进行讨论的，并且特指的是讨论简单函数闭包（见第 2 章的“保持作用域”）和简单对象（键值对的集合）。

一个简单的函数闭包：

```
function outer() {
    var one = 1;
    var two = 2;

    return function inner(){
        return one + two;
    };
}

var three = outer();

three();            // 3
```

一个简单的对象：

```
var obj = {
    one: 1,
    two: 2
};

function three(outer) {
    return outer.one + outer.two;
}

three( obj );        // 3
```

但提到“闭包“时，很多人会想很多额外的事情，例如异步回调甚至是封装和信息隐藏的模块模式。同样，”对象“会让人想起类、 `this` 、原型和大量其它的工具和模式。

随着深入，我们会需要小心地处理部分额外的相关内容，但是现在，尽量只记住闭包和对象最简单的释义 —— 这会减少很多探索过程中的困惑。

相像

闭包和对象之间的关系可能不是那么明显。让我们先来探究它们之间的相似点。

为了给这次讨论一个基调，让我简述两件事：

1. 一个没有闭包的编程语言可以用对象来模拟闭包。
2. 一个没有对象的编程语言可以用闭包来模拟对象。

换句话说，我们可以认为闭包和对象是一样东西的两种表达方式。

状态

思考下面的代码：

```
function outer() {
    var one = 1;
    var two = 2;

    return function inner(){
        return one + two;
    };
}

var obj = {
    one: 1,
    two: 2
};
```

`inner()` 和 `obj` 对象持有的作用域都包含了两个元素状态：值为 `1` 的 `one` 和值为 `2` 的 `two` 。从语法和机制来说，这两种声明状态是不同的。但概念上，他们的确相当相似。

事实上，表达一个对象为闭包形式，或闭包为对象形式是相当简单的。接下来，尝试一下：

```
var point = {
    x: 10,
    y: 12,
    z: 14
};
```

你是不是想起了一些相似的东西？

```
function outer() {
    var x = 10;
    var y = 12;
    var z = 14;

    return function inner(){
        return [x,y,z];
    }
};

var point = outer();
```

注意：每次被调用时 `inner()` 方法创建并返回了一个新的数组（亦然是一个对象）。这是因为 JS 不提供返回多个数据却不包装在一个对象中的能力。这并不是严格意义上的一个违反我们对象类似闭包的说明的任务，因为这只是一个暴露／运输具体值的实现，状态追踪本身仍然是基于对象的。使用 ES6+ 数组解构，我们可以声明地忽视这个临时中间对象通过另一种方式：`var [x,y,z] = point()`。从开发者工程学角度，值应该被单独存储并且通过闭包而不是对象来追踪。

如果你有一个嵌套对象会怎么样？

```
var person = {
    name: "Kyle Simpson",
    address: {
        street: "123 Easy St",
        city: "JS'ville",
        state: "ES"
    }
};
```

我们可以用嵌套闭包来表示相同的状态：

```
function outer() {
    var name = "Kyle Simpson";
    return middle();

    // ********************

    function middle() {
        var street = "123 Easy St";
        var city = "JS'ville";
        var state = "ES";

        return function inner(){
            return [name,street,city,state];
        };
    }
}

var person = outer();
```

让我们尝试另一个方向，从闭包转为对象：

```
function point(x1,y1) {
    return function distFromPoint(x2,y2){
        return Math.sqrt(
            Math.pow( x2 - x1, 2 ) +
            Math.pow( y2 - y1, 2 )
        );
    };
}

var pointDistance = point( 1, 1 );

pointDistance( 4, 5 );        // 5
```

`distFromPoint(..)` 封装了 `x1` 和 `y1`，但是我们也可以通过传入一个具体的对象作为替代值：

```
function pointDistance(point,x2,y2) {
    return Math.sqrt(
        Math.pow( x2 - point.x1, 2 ) +
        Math.pow( y2 - point.y1, 2 )
    );
};

pointDistance(
    { x1: 1, y1: 1 },
    4,    // x2
    5    // y2
);
// 5
```

明确地传入 `point` 对象替换了闭包的隐式状态。

行为，也是一样！

对象和闭包不仅是表达状态集合的方式，而且他们也可以包含函数或者方法。将数据和行为捆绑为有一个充满想象力的名字：封装。

思考：

```
function person(name,age) {
    return happyBirthday(){
        age++;
        console.log(
            "Happy " + age + "th Birthday, " + name + "!"
        );
    }
}

var birthdayBoy = person( "Kyle", 36 );

birthdayBoy();           // Happy 37th Birthday, Kyle!
```

内部函数 `happyBirthday()` 封闭了 `name` 和 `age` ，所以内部的函数也持有了这个状态。

我们也可以通过 this 绑定一个对象来获取同样的能力：

```
var birthdayBoy = {
    name: "Kyle",
    age: 36,
    happyBirthday() {
        this.age++;
        console.log(
            "Happy " + this.age + "th Birthday, " + this.name + "!"
        );
    }
};

birthdayBoy.happyBirthday();
// Happy 37th Birthday, Kyle!
```

我们仍然通过 `happyBrithday()` 函数来表达对状态数据的封装，但是用对象代替了闭包。同时我们没有显式给函数传递一个对象（如同先前的例子）；JavaScript 的 `this` 绑定可以创造一个隐式的绑定。

从另一方面分析这种关系：闭包将单个函数与一系列状态结合起来，而对象却在保有相同状态的基础上，允许任意数量的函数来操作这些状态。

事实上，我们可以在一个作为接口的闭包上将一系列的方法暴露出来。思考一个包含了两个方法的传统对象：

```
var person = {
    firstName: "Kyle",
    lastName: "Simpson",
    first() {
        return this.firstName;
    },
    last() {
        return this.lastName;
    }
}

person.first() + " " + person.last();
// Kyle Simpson
```

只用闭包而不用对象，我们可以表达这个程序为：

```
function createPerson(firstName,lastName) {
    return API;

    // ********************

    function API(methodName) {
        switch (methodName) {
            case "first":
                return first();
                break;
            case "last":
                return last();
                break;
        };
    }

    function first() {
        return firstName;
    }

    function last() {
        return lastName;
    }
}

var person = createPerson( "Kyle", "Simpson" );

person( "first" ) + " " + person( "last" );
// Kyle Simpson
```

尽管这些程序看起来感觉有点反人类，但它们实际上只是相同程序的不同实现。

（不）可变

许多人最初都认为闭包和对象行为的差别源于可变性；闭包会阻止来自外部的变化而对象则不然。但是，结果是，这两种形式都有典型的可变行为。

正如第 6 章讨论的，这是因为我们关心的是值的可变性，值可变是值本身的特性，不在于在哪里或者如何被赋值的。

```
function outer() {
    var x = 1;
    var y = [2,3];

    return function inner(){
        return [ x, y[0], y[1] ];
    };
}

var xyPublic = {
    x: 1,
    y: [2,3]
};
```

在 `outer()` 中字面变量 `x` 存储的值是不可变的 —— 记住，定义的基本类型如 `2` 是不可变的。但是 `y` 的引用值，一个数组，绝对是可变的。这点对于 `xyPublic` 中的 `x` 和 `y` 属性也是完全相同的。

通过指出 `y` 本身是个数组我们可以强调对象和闭包在可变这点上没有关系，因此我们需要将这个例子继续拆解：

```
function outer() {
    var x = 1;
    return middle();

    // ********************

    function middle() {
        var y0 = 2;
        var y1 = 3;

        return function inner(){
            return [ x, y0, y1 ];
        };
    }
}

var xyPublic = {
    x: 1,
    y: {
        0: 2,
        1: 3
    }
};
```

如果你认为这个如同“世界是一只驮着一只一直驮下去的乌龟（对象）群”，在最底层，所有的状态数据都是基本类型，而所有基本类型都是不可变值。

不论是用嵌套对象还是嵌套闭包代表状态，这些被持有的值都是不可变的。

同构

同构这个概念最近在 JavaScript 圈经常被提出，它通常被用来指代码可以同时被服务端和浏览器端使用／分享。我不久以前写了一篇博文说明这种对同构这个词的使用是错误的，隐藏了它实际上确切和重要的意思。

这里我是博文部分的节选：

> 同构的意思是什么？当然，我们可以用数学词汇，社会学或者生物学讨论它。同构最普遍的概念是你有两个类似但是不相同的结构。
>
> 在这些所有的惯用法中，同构和相等的区别在这里：如果两个值在各方面完全一致那么它们相等，但是如果它们表现不一致却仍有一对一或者双向映射的关系那么它们是同构。
>
> 换而言之，两件事物A和B如果你能够映射（转化）A 到 B 并且能够通过反向映射回到A那么它们就是同构。

回想第 2 章的简单数学回顾，我们讨论了函数的数学定义是一个输入和输出之间的映射。我们指出这在学术上称为态射。同构是双映（双向）态射的特殊案例，它需要映射不仅仅必须可以从任意一边完成，而且在任一方式下反应完全一致。

不去思考这些关于数字的问题，让我们将同构关联到代码。再一次引用我的博文：

> 如果 JS 有同构的话是怎么样的？它可能是一集合的 JS 代码转化为了另一集合的 JS 代码，并且（重要的是）如果你原意的话，你可以把转化后的代码转为之前的。

正如我们之前通过闭包如同对象和对象如同闭包为例声称的一样，它们的表达可以任意替换。就这一点来说，它们互为同构。

简而言之，闭包和对象是状态的同构表示（及其相关功能）。

下次你听到谁说 “X 与 Y 是同构的”，他们的意思是，“X 和 Y 可以从两者中的任意一方转化到另一方，并且无论怎样都保持了相同的特性。”

内部结构

所以，我们可以从我们写的代码角度想象对象是闭包的一种同构展示。但我们也可以观察到闭包系统可以被实现，并且很可能是用对象实现的！

这样想一下：在如下的代码中， 在 `outer()` 已经运行后，JS 如何为了 `inner()` 的引用保持对变量 `x` 的追踪？

```
function outer() {
    var x = 1;

    return function inner(){
        return x;
    };
}
```

我们会想到作用域， `outer()` 作为属性的对象实施设置所有的变量定义。因此，从概念上讲，在内存中的某个地方，是类似这样的。

```
scopeOfOuter = {
    x: 1
};
```

接下来对于 `inner()` 函数，一旦创建，它获得了一个叫做 `scopeOfInner` 的（空）作用域对象，这个对象被其 `[[Prototype]]` 连接到 `scopeOfOuter` 对象，近似这个：

```
scopeOfInner = {};
Object.setPrototypeOf( scopeOfInner, scopeOfOuter );
```

接着，当内部的 `inner()` 建立词法变量 `x` 的引用时，实际更像这样：

```
return scopeOfInner.x;
```

`scopeOfInner` 并没有一个 `x` 的属性，当他的 `[[Prototype]]` 连接到拥有 `x` 属性的 `scopeOfOuter` 时。通过原型委托访问 `scopeOfOuter.x` 返回值是 `1`。

这样，我们可以近似认为为什么 `outer()` 的作用域甚至在当它执行完都被保留（通过闭包），这是因为 `scopeOfInner` 对象连接到 `scopeOfOuter` 对象，因此，使这个对象和它的属性完整的被保存下来。

现在，这都只是概念。我没有从字面上说 JS 引擎使用对象和原型。但它完全有道理，它可以同样地工作。

许多语言实际上通过对象实现了闭包。另一些语言用闭包的概念实现了对象。但我们让读者使用他们的想象力思考这是如何工作的。

同根异枝

所以闭包和对象是等价的，对吗？不完全是，我打赌它们比你在读本章前想的更加相似，但是它们仍有重要的区别点。

这些区别点不应当被视作缺点或者不利于使用的论点；这是错误的观点。对于给定的任务，它们应该被视为使一个或另一个更适合（和可读）的特点和优势。

结构可变性

从概念上讲，闭包的结构不是可变的。

换而言之，你永远不能从闭包添加或移除状态。闭包是一个表示对象在哪里声明的特性（被固定在编写／编译时间），并且不受任何条件的影响 —— 当然假设你使用严格模式并且／或者没有使用作弊手段例如 `eval(..)`。

注意：JS 引擎可以从技术上过滤一个对象来清除其作用域中不再被使用的变量，但是这是一个对于开发者透明的高级的优化。无论引擎是否实际做了这类优化，我认为对于开发者来说假设闭包是作用域优先而不是变量优先是最安全的。如果你不想保留它，就不要封闭它（在闭包里）！

但是，对象默认是完全可变的，你可以自由的添加或者移除（`delete`）一个对象的属性／索引，只要对象没有被冻结（`Object.freeze(..)`）

这或许是代码可以根据程序中运行时条件追踪更多（或更少）状态的优势。

举个例子，让我们思考追踪游戏中的按键事件。几乎可以肯定，你会考虑使用一个数组来做这件事：

```
function trackEvent(evt,keypresses = []) {
    return keypresses.concat( evt );
}

var keypresses = trackEvent( newEvent1 );

keypresses = trackEvent( newEvent2, keypresses );
```

注意：你能否认出为什么我使用 `concat(..)` 而不是直接对 `keypresses` 数组使用 `push(..)` 操作？因为在函数式编程中，我们通常希望对待数组如同不可变数据结构，可以被创建和添加，但不能直接改变。我们剔除了显式重新赋值带来的邪恶副作用（稍后再作说明）。

尽管我们不在改变数组的结构，但当我们希望时我们也可以。稍后详细介绍。

数组不是记录这个 evt 对象的增长“列表”的仅有的方式。。我们可以使用闭包：

```
function trackEvent(evt,keypresses = () => []) {
    return function newKeypresses() {
        return [ ...keypresses(), evt ];
    };
}

var keypresses = trackEvent( newEvent1 );

keypresses = trackEvent( newEvent2, keypresses );
```

你看出这里发生了什么吗？

每次我们添加一个新的事件到这个“列表”，我们创建了一个包装了现有 `keypresses()` 方法（闭包）的新闭包，这个新闭包捕获了当前的 `evt` 。当我们调用 `keypresses()` 函数，它将成功地调用所有的内部方法，并创建一个包含所有独立封装的 `evt` 对象的中间数组。再次说明，闭包是一个追踪所有状态的机制；这个你看到的数组只是一个对于需要一个方法来返回函数中多个值的具体实现。

所以哪一个更适合我们的任务？毫无意外，数组方法可能更合适一些。闭包的不可变结构意味着我们的唯一选项是封装更多的闭包在里面。对象默认是可扩展的，所以我们需要增长这个数组就足够了。

顺便一提，尽管我们表现出结构不可变或可变是一个闭包和对象之间的明显区别，然而我们使用对象作为一个不可变数据的方法实际上使之更相似而非不同。

数组每次添加就创造一个新数组（通过 `concat(..)` ）就是把数组对待为结构不可变，这个概念上对等于通过适当的设计使闭包结构上不可变。

私有

当对比分析闭包和对象时可能你思考的第一个区分点就是闭包通过词法作用域提供“私有”状态，而对象将一切做为公共属性暴露。这种私有有一个精致的名字：信息隐藏。

考虑词法闭包隐藏：

```
function outer() {
    var x = 1;

    return function inner(){
        return x;
    };
}

var xHidden = outer();

xHidden();              // 1
```

现在同样的状态公开：

```
var xPublic = {
    x: 1
};

xPublic.x;              // 1
```

这里有一些在常规的软件工程原理方面明显的区别 —— 考虑下抽象，这种模块模式有着公有和私有 API 等等。但是让我们试着把我们的讨论局限于函数式编程的观点，毕竟，这是一本关于函数式编程的书！

可见性

似乎隐藏信息的能力是一种理想状态的跟踪特性，但是我认为函数式编程者可能持反对观点。

在一个对象中管理状态作为公开属性的一个优点是这使你状态中的所有数据更容易枚举（迭代）。思考下你想访问每一个按键事件（从之前的那个例子）并且存储到一个数据库，使用一个这样的工具：

```
function recordKeypress(keypressEvt) {
    // 数据库实用程序
    DB.store( "keypress-events", keypressEvt );
}
```

If you already have an array -- just an object with public numerically-named properties -- this is very straightforward using a built-in JS array utility `forEach(..)`:

如果你已经有一个数组，正好是一个拥有公开的用数字命名属性的对象 —— 非常直接地使用 JS 对象的内建工具 `forEach(..)`：

```
keypresses.forEach( recordKeypress );
```

但是，如果按键列表被隐藏在一个闭包里，你不得不在闭包内暴露一个享有特权访问数据的公开 API 工具。

举例而说，我可以给我们的闭包 —— keypresses 例子自有的 forEach 方法，如同数组内建的：

```
function trackEvent(
    evt,
    keypresses = {
        list() { return []; },
        forEach() {}
    }
) {
    return {
        list() {
            return [ ...keypresses.list(), evt ];
        },
        forEach(fn) {
            keypresses.forEach( fn );
            fn( evt );
        }
    };
}

// ..

keypresses.list();        // [ evt, evt, .. ]

keypresses.forEach( recordKeypress );
```

对象状态数据的可见性让我们能更直接地使用它，而闭包遮掩状态让我们更艰难地处理它。

变更控制

如果词法变量被隐藏在一个闭包中，只有闭包内部的代码才能自由的重新赋值，在外部修改 `x` 是不可能的。

正如我们在第 6 章看到的，提升代码可读性的唯一真相就是减少表面掩盖，读者必须可以预见到每一个给定变量的行为。

词法（作用域）在重新赋值上的局部就近原则是为什么我不认为 const 是一个有帮助的特性的一个重要原因。作用域（例如闭包）通常应该尽可能小，这意味着重新赋值只会影响少许代码。在上面的 `outer()` 中，我们可以快速地检查到没有一行代码重设了 x，至此（x 的）所有意图和目的表现地像一个常量。

这类保证对于我们对函数纯净的信任是一个强有力的贡献，例如。

换而言之，`xPublic.x` 是一个公开属性，程序的任何部分都能引用 `xPublic` ，默认有重设 `xPublic.x` 到别的值的能力。这会让很多行代码需要被考虑。

这是为什么在第 6 章， 我们视 `Object.freeze(..)` 为使所有的对象属性只读（writable: false）的一个快速而凌乱的方式，让它们不能被不可预测的重设。

不幸的是，`Object.freeze(..)` 是极端且不可逆的。

使有了闭包，你就有了一些可以更改代码的权限，而剩余的程序是受限的。当我们冻结一个对象，代码中没有任何部分可以被重设。此外，一旦一个对象被冻结，它不能被解冻，所以所有属性在程序运行期间都保持只读。

在我想允许重新赋值但是在表层限制的地方，闭包比起对象更方便和灵活。在我不想重新赋值的地方，一个冻结的对象比起重复 `const` 声明在我所有的函数中更方便一些。

许多函数式编程者在重新赋值上采取了一个强硬的立场：它不应该被使用。他们倾向使用 `const` 来使用所有闭包变量只读，并且他们使用 `Ojbect.freeze(..)` 或者完全不可变数据结构来防止属性被重新赋值。此外，他们尽量在每个可能的地方减少显式地声明的／追踪的变量，更倾向于值传递 —— 函数链，作为参数被传递的 `return` 值，等等 —— 替代中间值存储。

这本书是关于 JavaScript 中的轻量级函数式编程，这是一个我与核心函数式编程群体有分歧的情况。

我认为变量重新赋值当被合理的使用时是相当有用的，它的明确性具有相当有可读性。从经验来看，在插入 `debugger` 或断点或跟踪表表达式时，调试工作要容易得多。

状态拷贝

正如我们在第 6 章学习的，防止副作用侵蚀代码可预测性的最好方法之一是确保我们将所有状态值视为不可变的，无论他们是否真的可变（冻结）与否。

如果你没有使用特别定制的库来提供复杂的不可变数据结构，最简单满足要求的方法：在每次变化前复制你的对象或者数组。

数组浅拷贝很容易：只要使用 `slice()` 方法：

```
var a = [ 1, 2, 3 ];

var b = a.slice();
b.push( 4 );

a;              // [1,2,3]
b;              // [1,2,3,4]
```

对象也可以相对容易地实现浅拷贝：

```
var o = {
    x: 1,
    y: 2
};

// 在 ES2017 以后，使用对象的解构：
var p = { ...o };
p.y = 3;

// 在 ES2015 以后：
var p = Object.assign( {}, o );
p.y = 3;
```

如果对象或数组中的值是非基本类型（对象或数组），使用深拷贝你不得不手动遍历每一层来拷贝每个内嵌对象。否则，你将有这些内部对象的共享引用拷贝，这就像给你的程序逻辑造成了一次大破坏。

你是否意识到克隆是可行的只是因为所有的这些状态值是可见的并且可以如此简单地被拷贝？一堆被包装在闭包里的状态会怎么样，你如何拷贝这些状态？

那是相当乏味的。基本上，你不得不做一些类似之前我们自定义 `forEach` API 的方法：提供一个闭包内层拥有提取或拷贝隐藏值权限的函数，并在这过程中创建新的等价闭包。

尽管这在理论上是可行的，对读者来说也是一种锻炼！这个实现的操作量远远不及你可能进行的任何真实程序的调整。

在表示需要拷贝的状态时，对象具有一个更明显的优势。

性能

从实现的角度看，对象有一个比闭包有利的原因，那就是 JavaScript 对象通常在内存和甚至计算角度是更加轻量的。

但是需要小心这个普遍的断言：有很多东西可以用来处理对象，这会抹除你从无视闭包转向对象状态追踪获得的任何性能增益。

让我们考虑一个情景的两种实现。首先，闭包方式实现：

```
function StudentRecord(name,major,gpa) {
    return function printStudent(){
        return `${name}, Major: ${major}, GPA: ${gpa.toFixed(1)}`;
    };
}

var student = StudentRecord( "Kyle Simpson", "kyle@some.tld", "CS", 4 );

// 随后

student();
// Kyle Simpson, Major: CS, GPA: 4.0
```

内部函数 `printStudent()` 封装了三个变量：`name`、`major` 和 `gpa`。它维护这个状态无论我们是否传递引用给这个函数，在这个例子我们称它为 `student()`。

现在看对象（和 `this`）方式：

```
function StudentRecord(){
    return `${this.name}, Major: ${this.major}, GPA: ${this.gpa.toFixed(1)}`;
}

var student = StudentRecord.bind( {
    name: "Kyle Simpson",
    major: "CS",
    gpa: 4
} );

// 随后

student();
// Kyle Simpson, Major: CS, GPA: 4.0
```

`student()` 函数，学术上叫做"边界函数"——有一个硬性边界 `this` 来引用我们传入的对象字面量，因此之后任何调用 `student()` 将使用这个对象作为 `this`，于是它的封装状态可以被访问。

两种实现有相同的输出：一个保存状态的函数，但是关于性能，会有什么不同呢？

注意：精准可控地判断 JS 代码片段性能是非常困难的事情。我们在这里不会深入所有的细节，但是我强烈推荐你阅读《你不知道的 JS：异步和性能》这本书，特别是第 6 章"性能测试和调优"，来了解细节。

如果你写过一个库来创造持有配对状态的函数，要么在第一个片段中调用 `studentRecord(..)`，要么在第二个片段中调用 `StudentRecord.bind(..)` 的方式，你可能更多的关心它们两的性能怎样。检查代码，我们可以看到前者每次都必须创建一个新函数表达

式。后者使用 `bind(..)`，没有明显的含义。

思考 `bind(..)` 在内部做了什么的一种方式是创建一个闭包来替代函数，像这样：

```
function bind(orinFn,thisObj) {
    return function boundFn(...args) {
        return origFn.apply( thisObj, args );
    };
}

var student = bind( StudentRecord, { name: "Kyle.." } );
```

这样，看起来我们的场景的两种实现都是创造一个闭包，所以性能看似也是一致的。

但是，内置的 `bind(..)` 工具并不一定要创建闭包来完成任务。它只是简单地创建了一个函数，然后手动设置它的内部 `this` 给一个指定的对象。这可能比起我们使用闭包本身是一个更高效的操作。

我们这里讨论的在每次操作上的这种性能优化是不值一提的。但是如果你的库的关键部分被使用了成千上万次甚至更多，那么节省的时间会很快增加。许多库 —— Bluebird 就是这样一个例子，它已经完成移除闭包去使用对象的优化。

在库的使用案例之外，持有配对状态的函数通常在应用的关键路径发生的次数相对非常少。相比之下，典型的使用是函数加状态 —— 在任意一个片段调用 `student()`，是更加常见的。

如果你的代码中也有这样的场景，你应该更多地考虑（优化）前后的性能对比。

历史上的边界函数通常具有一个相当糟糕的性能，但是最近已经被 JS 引擎高度优化。如果你在几年前检测过这些变化，很可能跟你现在用最近的引擎重复测试的结果完全不一致。

边界函数现在看起来至少跟同样的封装函数表现的一样好。所以这是另一个支持对象比闭包好的点。

我只想重申：性能观察结果不是绝对的，在一个给定场景下决定什么是最好的是非常复杂的。不要随意使用你从别人那里听到的或者是你从之前一些项目中看到的。小心的决定对象还是闭包更适合这个任务。

总结

本章的真理无法被直述。必须阅读本章来寻找它的真理。

JavaScript 轻量级函数式编程

第 8 章：列表操作

你是否还沉迷于上一节介绍的闭包／对象之中？欢迎回来！

> 如果你能做一些令人惊叹的事情，请持续保持下去。

本文之前已经简要的提及了一些实用函数：`map(..)`、`filter(..)` 和 `reduce(..)`，现在深入了解一下它们。在 Javascript 中，这些实用函数通常被用于 Array（即 “list” ）的原型上。因此可以很自然的将这些实用函数和数组或列表操作联系起来。

在讨论具体的数组方法之前，我们应该很清楚这些操作的作用。在这章中，弄明白为何有这些列表操作和这些操作如何工作同等重要。请保持头脑清晰，跟上节奏。

在本章内外，有大量常见且通俗易懂的列表操作的例子，它们描述一些细小的操作去处理一系列的值（如数组中的每一个值加倍）。这样通俗易懂。

但是不要停留在这些简单示例的表面，而错过了更深层次的点。通过对一系列任务建模来理解一些非常重要的函数式编程在列表操作中的价值 —— 一些些看起来不像列表的语句 —— 作为列表操作，而不是单独执行。

这不仅仅是编写许多简练代码的技巧。我们所要做的是，从命令式转变为声明式风格，使代码模式更容易辨认，从而可读性更好。

但这里有一些更需要掌握的东西。在命令式代码中，一组计算的中间结果都是通过赋值来存储。代码中依赖的命令模式越多，越难验证它们不是错误。比如，在逻辑上，值的意外改变，或隐藏的潜在原因／影响。

通过与／或链接组合列表操作，中间结果被隐式地跟踪，并在很大程度上避免了这些风险。

注意： 相比前面几章，为了代码片段更加简练，我们将采用 ES6 的箭头函数。尽管第 2 章中对于箭头函数的建议依旧普遍适用于编码中。

非函数式编程列表处理

作为本章讨论的快速预览，我想调用一些操作，这些操作看上去可以将 Javascript 数组和函数式编程列表操作相关联，但事实上并没有。我们不会在这里讨论这些，因为它们与一般的函数式编程最佳实践不一致：

- `forEach(..)`

- `some(..)`
- `every(..)`

`forEach(..)` 是遍历辅助函数，但是它被设计为带有副作用的函数来处理每次遍历；你或许已经猜测到了它为什么不是我们正在讨论的函数式编程列表操作！

`some(..)` 和 `every(..)` 鼓励使用纯函数（具体来说，就像 `filter(..)` 这样的谓词函数），但是它们不可避免地将列表化简为 `true` 或 `false` 的值，本质上就像搜索和匹配。这两个实用函数和我们期望采用函数式编程来组织代码相匹配，因此，这里我们将跳过它们。

映射

我们将采用最基础和最简单的操作 `map(..)` 来开启函数式编程列表操作的探索。

映射的作用就将一个值转换为另一个值。例如，如果你将 `2` 乘以 `3`，你将得到转换的结果 `6` 。需要重点注意的是，我们并不是在讨论映射转换是暗示就地转换或重新赋值，而是将一个值从一个地方映射到另一个新的地方。

换句话说

```
var x = 2, y;

// 转换／投影
y = x * 3;

// 变换／重新赋值
x = x * 3;
```

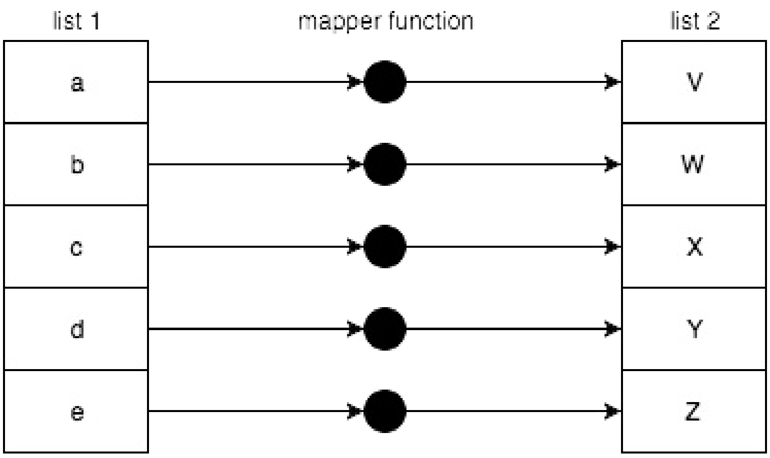
如果我们定义了乘 `3` 这样的函数，这个函数充当映射（转换）的功能。

```
var multipleBy3 = v => v * 3;

var x = 2, y;

// 转换／投影
y = multiplyBy3( x );
```

我们可以自然的将映射的概念从单个值扩展到值的集合。`map(..)` 操作将列表中所有的值转换为新列表中的列表项，如下图所示：



实现 `map(..)` 的代码如下：

```
function map(mapperFn,arr) {
    var newList = [];

    for (let idx = 0; idx < arr.length; idx++) {
        newList.push(
            mapperFn( arr[idx], idx, arr )
        );
    }

    return newList;
}
```

注意： `mapperFn, arr` 的参数顺序，乍一看像是在倒退。但是这种方式在函数式编程类库中非常常见。因为这样做，可以让这些实用函数更容易被组合。

`mapperFn(..)` 自然地将传入的列表项做映射/转换，并且也传入了 `idx` 和 `arr` 。这样做，可以和内置的数组的 `map(..)` 保持一致。在某些情况下，这些额外的参数非常有用。

但是，在一些其他情况中，你只希望传递列表项到 `mapperFn(..)` 。因为额外的参数可能会改变它的行为。在第三章的"共同目的（ All for one ）"中，我们介绍了 unary(..)，它限制函数仅仅接受一个参数，不论多少个参数被传入。

回顾第三章关于把 `parseInt()` 的参数数量限制为 1，从而使之成为可被安全使用的 `mapperFn()` 的例子：

```
map( ["1","2","3"], unary( parseInt ) );
// [1,2,3]
```

Javascript 提供了内置的数组操作方法 `map(..)` ，这个方法使得列表中的链式操作更为便利。

注意：Javascript 数组中的原型中定义的操作(`map(..)` 、 `filter(..)` 和 `reduce(..)`)的最后一个可选参数可以被用于绑定 “this” 到当前函数。我们在第二章中曾经讨论过“什么是 this？”，以及在函数式编程的最佳实践中应该避免使用 `this` 。基于这个原因，在这章中的示例中，我们不采用 `this` 绑定功能。

除了明显的字符和数字操作外，你可以对列表中的这些值类型进行操作。我们可以采用 `map(..)` 方法来通过函数列表转换得到这些函数返回的值，示例代码如下:

```
var one = () => 1;
var two = () => 2;
var three = () => 3;

[one,two,three].map( fn => fn() );
// [1,2,3]
```

我们也可以先将函数放在列表中，然后组合列表中的每一个函数，最后执行它们，代码如下：

```
var increment = v => ++v;
var decrement = v => --v;
var square = v => v * v;

var double = v => v * 2;

[increment,decrement,square]
.map( fn => compose( fn, double ) )
.map( fn => fn( 3 ) );
// [7,5,36]
```

我们注意到关于 `map(..)` 的一些有趣的事情：我们通常假定列表是从左往右执行的，但 `map(..)` 没有这个概念，它确实不需要这个次序。每一个转换应该独立于其他的转换。

映射普遍适用于并行处理的场景中，尤其在处理大列表时可以提升性能。但是在 Javascript 中，我们并没有看到这样的场景。因为这里不需要你传入诸如 `mapperFn(..)` 这样的纯函数，即便你应当这样做。如果传入了非纯函数，JS 在不同的顺序中执行不同的方法，这将很快产生大问题。

尽管从理论上讲，单个映射操作是独立的，但 JS 需要假定它们不是。这是令人讨厌的。

同步 vs 异步

这篇文章中讨论的列表操作都是同步地操作一组已经存在的值组成的列表， `map(..)` 在这里被看作是急切的操作。但另外一种思考方式是将映射函数作为时间处理器，该处理器会在新元素加入到列表中时执行。

想象一下这样的场景：

```
var newArr = arr.map();

arr.addEventListener( "value", multiplyBy3 );
```

现在，任何时候，当一个值加入到 `arr` 中的时候，`multiplyBy3(..)` 事件处理器（映射函数）将加入的值当参数执行，将转换后的结果加入到 `newArr` 。

我们建议，数组以及在数组上应用的数组操作都是迫切的同步的，然而，这些相同的操作也可以应用在一直接受新值的“惰性列表”（即流）上。我们将在第 10 章中深入讨论它。

映射 vs 遍历

有些人提倡在迭代的时候采用 `map(..)` 替代 `forEach(..)` ，它本质上不会去触碰接受到的值，但仍有可能产生副作用：

```
[1,2,3,4,5]
.map( function mapperFn(v){
    console.log( v );           // 副作用!
    return v;
} )
..
```

这种技术似乎非常有用的原因是 `map(..)` 返回数组，这样你可以在它之后继续链式执行更多的操作。而 `forEach(..)` 返回的的值是 `undefined` 。然而，我认为你应当避免采用这种方式使用 `map(..)` ，因为这里明显的以非函数式编程的方式使用核心的函数式编程操作，将引起巨大的困惑。

你应该听过一句老话，用合适的工具做合适的事，对吗？锤子敲钉子，螺丝刀拧螺丝等等。这里有些细微的不同：采用恰当的方式使用合适的工具。

锤子是挥动手敲的，如果你尝试采用嘴去钉钉子，效率会大打折扣。 `map(..)` 是用来映射值的，而不是带来副作用。

一个词：函子

在这本书中，我们尽可能避免使用人为创造的函数式编程术语。我们有时候会使用官方术语，但在大多数时候，采用日常用语来描述更加通俗易懂。

这里我将被一个可能会引起恐慌的词：函子来短暂地打断这种通俗易懂的模式。这里之所以要讨论函子的原因是我们已经了解了它是干什么的，并且这个词在函数式编程文献中被大量使用。你不会被这个词吓到而带来副作用。

函子是采用运算函数有效用操作的值。

如果问题中的值是复合的，意味着它是由单个值组成，就像数组中的情况一样。例如，函子在每个单独的值上执行操作函数。函子实用函数创建的新值是所有单个操作函数执行的结果的组合。

这就是用 `map(..)` 来描述我们所看到东西的一种奇特方式。`map(..)` 函数采用关联值（数组）和映射函数（操作函数），并为数组中的每一个独立元素执行映射函数。最后，它返回由所有新映射值组成的新数组。

另一个例子：字符串函子是一个字符串加上一个实用函数，这个实用函数在字符串的所有字符上执行某些函数操作，返回包含处理过的字符的字符串。参考如下非常刻意的例子：

```
function uppercaseLetter(c) {
    var code = c.charCodeAt( 0 );

    // 小写字母?
    if (code >= 97 && code <= 122) {
        // 转换为大写!
        code = code - 32;
    }

    return String.fromCharCode( code );
}

function stringMap(mapperFn,str) {
    return [...str].map( mapperFn ).join( "" );
}

stringMap( uppercaseLetter, "Hello World!" );
// 你好，世界!
```

`stringMap(..)` 允许字符串作为函子。你可以定义一个映射函数用于任何数据类型。只要实用函数满足这些规则，该数据结构就是一个函子。

过滤器

想象一下，我带着空篮子去逛食品杂货店的水果区。这里有很多水果（苹果、橙子和香蕉）。我真的很饿，因此我想要尽可能多的水果，但是我真的更喜欢圆形的水果（苹果和橙子）。因此我逐一筛选每一个水果，然后带着装满苹果和橙子的篮子离开。

我们将这个筛选的过程称为“过滤”。将这次购物描述为从空篮子开始，然后只过滤（挑选，包含）出苹果和橙子，或者从所有的水果中过滤掉（跳过，不包括）香蕉。你认为哪种方式更自然？

如果你在一锅水里面做意大利面条，然后将这锅面条倒入滤网（过滤）中，你是过滤了意大利面条，还是过滤掉了水？如果你将咖啡渣放入过滤器中，然后泡一杯咖啡，你是将咖啡过滤到了杯子里，还是说将咖啡渣过滤掉？

你有没有发现过滤的结果取决于你想要把什么保留在过滤器中，还是说用过滤器将其过滤出去？

那么在航空／酒店网站上如何指定过滤选项呢？你是按照你的标准过滤结果，还是将不符合标准的过滤掉？仔细想想，这个例子也许和前面有不相同的语意。

取决于你的想法，过滤是排除的或者保留的，这种概念上的融合，使其难以理解。

我认为最通常的理解过滤（在编程之外）是剔除掉不需要的成员。不幸的是，在程序中我们基本上将这个语意倒转为更像是过滤需要的成员。

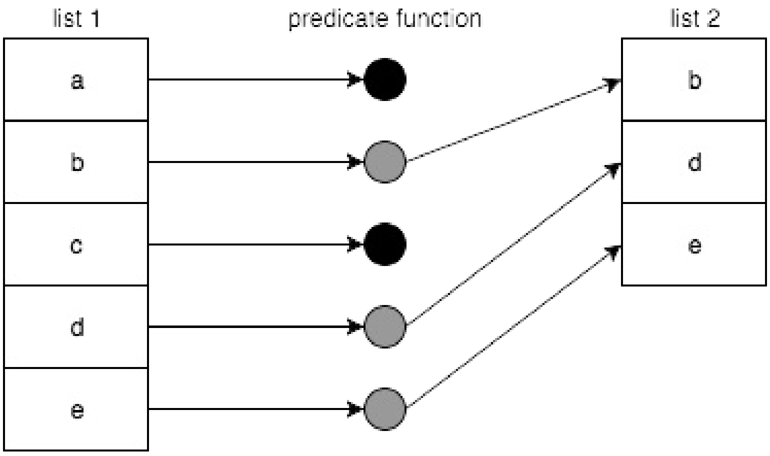
列表的 `filter(..)` 操作采用一个函数确定每一项在新数组中是保留还是剔除。这个函数返回 `true` 将保留这一项，返回 `false` 将剔除这一项。这种返回 `true` / `false` 来做决定的函数有一个特别的称谓：谓词函数。

如果你认为 `true` 是积极的信号，`filter(..)` 的定义是你是“保留”一个值，而不是“抛弃”一个值。

如果 `filter(..)` 被用于剔除操作，你需要转动你的脑子，积极的返回 `false` 发出排除的信号，并且被动的返回 `true` 来让一个值通过过滤器。

这种语意上不匹配的原因是你会将这个函数命名为 `predicateFn(..)`，这对于代码的可读性有意义，我们很快会讨论这一点。

下图很形象的介绍了列表间的 `filter(..)` 操作：



实现 `filter(..)` 的代码如下：

```js
function filter(predicateFn,arr) {
    var newList = [];

    for (let idx = 0; idx < arr.length; idx++) {
        if (predicateFn( arr[idx], idx, arr )) {
            newList.push( arr[idx] );
        }
    }

    return newList;
}
```

注意，就像之前的 `mapperFn(..)`，`predicateFn(..)` 不仅仅传入了值，还传入了 `idx` 和 `arr`。如果有必要，也可以采用 `unary(..)` 来限制它的形参。

正如 `map(..)`，`filter(..)` 也是 JS 数组内置支持的实用函数。

我们将谓词函数定义这样：

```js
var whatToCallIt = v => v % 2 == 1;
```

这个函数采用 `v % 2 == 1` 来返回 `true` 或 `false`。这里的效果是，值为奇数时返回 `true`，值为偶数时返回 `false`。这样，我们该如何命名这个函数？一个很自然的名字可能是：

```js
var isOdd = v => v % 2 == 1;
```

考虑一下如何在你的代码中使用 `isOdd(..)` 来做简单的值检查：

```js
var midIdx;

if (isOdd( list.length )) {
    midIdx = (list.length + 1) / 2;
}
else {
    midIdx = list.length / 2;
}
```

有感觉了，对吧？让我们采用内置的数组的 `filter(..)` 来对一组值做筛选：

```js
[1,2,3,4,5].filter( isOdd );
// [1,3,5]
```

如果让你描述 `[1,3,5]` 这个结果，你是说“我将偶数过滤掉了”，还是说“我做了奇数的筛选”？我认为前者是更自然的描述。但后者的代码可读性更好。阅读代码几乎是逐字的，这样我们“过滤的每一个数字都是奇数”。

我个人觉得这语意混乱。对于经验丰富的开发者来说，这里毫无疑问有大量的先例。但是对于一个新手来说，这个逻辑表达看上去不采用双重否定不好表达，换句话说，采用双重否定来表达比较好。

为了便以理解，我们可以将这个函数从 `isOdd(..)` 重命名为 `isEven(..)`：

```
var isEven = v => v % 2 == 1;

[1,2,3,4,5].filter( isEven );
// [1,3,5]
```

耶，但是这个函数名变得无意义，下面的示例中，传入的偶数，确返回了 `false`

```
isEven( 2 );        // false
```

呸！

回顾在第 3 章中的 "No Points"，我们定义 `not(..)` 操作来反转谓词函数，代码如下：

```
var isEven = not( isOdd );

isEven( 2 );        // true
```

但在前面定义的 `filter(..)` 方式中，无法使用这个 `isEven(..)`，因为它的逻辑已经反转了。我们将以偶数结束，而不是奇数，我们需要这么做：

```
[1,2,3,4,5].filter( not( isEven ) );
// [1,3,5]
```

这样完全违背了我们的初衷，所以我们不要这么做。这样，我们转一圈又回来了。

过滤掉 & 过滤

为了消除这些困惑，我们定义 `filterOut(..)` 函数来执行过滤掉那些值，而实际上其内部执行否定的谓词检查。这样，我们将已经定义的 `filter(..)` 设置别名为 `filterIn(..)`。

```
var filterIn = filter;

function filterOut(predicateFn,arr) {
    return filterIn( not( predicateFn ), arr );
}
```

现在，我们可以在任意过滤操作中，使用语意化的过滤器，代码如下所示：

```
isOdd( 3 );                              // true
isEven( 2 );                             // true

filterIn( isOdd, [1,2,3,4,5] );          // [1,3,5]
filterOut( isEven, [1,2,3,4,5] );        // [1,3,5]
```

我认为采用 `filterIn(..)` 和 `filterOut(..)`（在 Ramda 中称之为 `reject(..)`）会让代码的可读性比仅仅采用 `filter(..)` 更好。

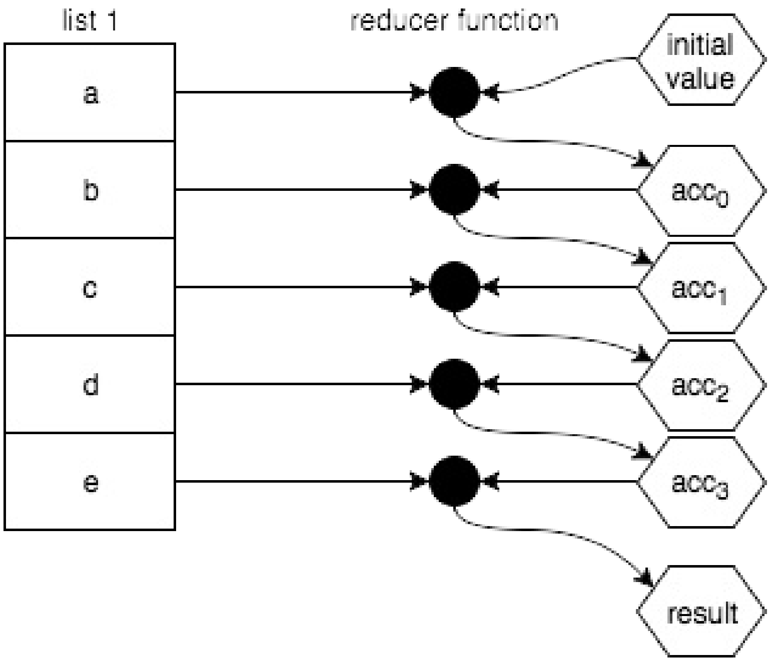
Reduce

`map(..)` 和 `filter(..)` 都会产生新的数组，而第三种操作（`reduce(..)`）则是典型地将列表中的值合并（或减少）到单个值（非列表），比如数字或者字符串。本章后续会探讨如何采用高级的方式使用 `reduce(..)`。`reduce(..)` 是函数式编程中的最重要的实用函数之一。就像瑞士军刀一样，具有丰富的用途。

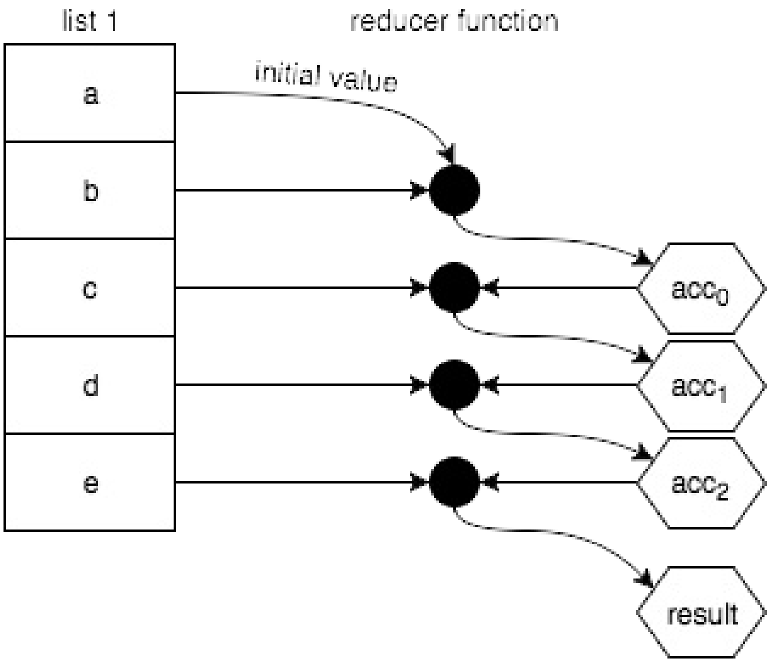
组合或缩减被抽象的定义为将两个值转换成一个值。有些函数式编程文献将其称为“折叠”，就像你将两个值合并到一个值。我认为这对于可视化是很有帮助的。

就像映射和过滤，合并的方式完全取决于你，一般取决于列表中值的类型。例如，数字通常采用算术计算合并，字符串采用拼接的方式合并，函数采用组合调用来合并。

有时候，缩减操作会指定一个 `initialValue`，然后将这个初始值和列表的第一个元素合并。然后逐一和列表中剩余的元素合并。如下图所示：



你也可以去掉上述的 `initialValue` ，直接将第一个列表元素当做 `initialValue` ，然后和列表中的第二个元素合并，如下图所示：



警告： 在 JavaScript 中，如果在缩减操作的列表中一个值都没有（在数组中，或没有指定 `initialValue` ），将会抛出异常。一个缩减操作的列表有可能为空的时候，需要小心采用不指定 `initialValue` 的方式。

传递给 `reduce(..)` 执行缩减操作的函数执行一般称为缩减器。缩减器和之前介绍的映射和谓词函数有不同的特征。缩减器主要接受当前的缩减结果和下一个值来做缩减操作。每一步缩减的当前结果通常称为累加器。

例如，对 5、10、15 采用初始值为 3 执行乘的缩减操作：

1. `3` * `5` = `15`
2. `15` * `10` = `150`
3. `150` * `15` = `2250`

在 JavaScript 中采用内置的 `reduce(..)` 方法来表达列表的缩减操作：

```
[5,10,15].reduce( (product,v) => product * v, 3 );
// 2250
```

我们可以采用下面的方式实现 `reduce(..)` ：

```
function reduce(reducerFn,initialValue,arr) {
    var acc, startIdx;

    if (arguments.length == 3) {
        acc = initialValue;
        startIdx = 0;
    }
    else if (arr.length > 0) {
        acc = arr[0];
        startIdx = 1;
    }
    else {
        throw new Error( "Must provide at least one value." );
    }

    for (let idx = startIdx; idx < arr.length; idx++) {
        acc = reducerFn( acc, arr[idx], idx, arr );
    }

    return acc;
}
```

就像 `map(..)` 和 `filter(..)` ，缩减函数也传递不常用的 `idx` 和 `arr` 形参，以防缩减操作需要。我不会经常用到它们，但我觉得保留它们是明智的。

在第 4 章中，我们讨论了 `compose(..)` 实用函数，和展示了用 `reduce(..)` 来实现的例子：

```
function compose(...fns) {
    return function composed(result){
        return fns.reverse().reduce( function reducer(result,fn){
            return fn( result );
        }, result );
    };
}
```

基于不同的组合，为了说明 `reduce(..)`，可以认为缩减器将函数从左到右组合（就像 `pipe(..)` 做的事情）。在列表中这样使用：

```
var pipeReducer = (composedFn,fn) => pipe( composedFn, fn );

var fn =
    [3,17,6,4]
    .map( v => n => v * n )
    .reduce( pipeReducer );

fn( 9 );            // 11016  (9 * 3 * 17 * 6 * 4)
fn( 10 );            // 12240  (10 * 3 * 17 * 6 * 4)
```

不幸的是，`pipeReducer(..)` 是非点自由的（见第 3 章中的“无形参”），但我们不能仅仅以缩减器本身来传递 `pipe(..)`，因为它是可变的；传递给 `reduce(..)` 额外的参数（`idx` 和 `arr`）会产生问题。

前面，我们讨论采用 `unary(..)` 来限制 `mapperFn(..)` 或 `predicateFn(..)` 仅采用一个参数。`binary(..)` 做了类似的事情，但在 `reducerFn(..)` 中限定两个参数：

```
var binary =
    fn =>
        (arg1,arg2) =>
            fn( arg1, arg2 );
```

采用 `binary(..)`，相比之前的示例有一些简洁：

```
var pipeReducer = binary( pipe );

var fn =
    [3,17,6,4]
    .map( v => n => v * n )
    .reduce( pipeReducer );

fn( 9 );            // 11016  (9 * 3 * 17 * 6 * 4)
fn( 10 );            // 12240  (10 * 3 * 17 * 6 * 4)
```

不像 `map(..)` 和 `filter(..)`，对传入数组的次序没有要求。`reduce(..)` 明确要采用从左到右的处理方式。如果你想从右到左缩减，JavaScript 提供了 `reduceRight(..)` 函数，它和 `reduce(..)` 的行为出了次序不一样外，其他都相同。

```
var hyphenate = (str,char) => str + "-" + char;

["a","b","c"].reduce( hyphenate );
// "a-b-c"

["a","b","c"].reduceRight( hyphenate );
// "c-b-a"
```

`reduce(..)` 采用从左到右的方式工作，很自然的联想到组合函数中的 `pipe(..)` 。 `reduceRight(..)` 从右往左的方式能自然的执行 `compose(..)` 。因此，我们重新采用 `reduceRight(..)` 实现 `compose(..)` ：

```
function compose(...fns) {
    return function composed(result){
        return fns.reduceRight( function reducer(result,fn){
            return fn( result );
        }, result );
    };
}
```

这样，我们不需要执行 `fns.reverse()` ；我们只需要从另一个方向执行缩减操作！

Map 也是 Reduce

`map(..)` 操作本质来说是迭代，因此，它也可以看作是（ `reduce(..)` ）操作。这个技巧是将 `reduce(..)` 的 `initialValue` 看成它自身的空数组。在这种情况下，缩减操作的结果是另一个列表！

```
var double = v => v * 2;

[1,2,3,4,5].map( double );
// [2,4,6,8,10]

[1,2,3,4,5].reduce(
    (list,v) => (
        list.push( double( v ) ),
        list
    ), []
);
// [2,4,6,8,10]
```

注意： 我们欺骗了这个缩减器，并允许采用 `list.push(..)` 去改变传入的列表所带来的副作用。一般来说，这并不是一个好主意，但我们清楚创建和传入 `[]` 列表，这样就不那么危险了。创建一个新的列表，并将 val 合并到这个列表的最后面。这样更有条理，并且性能开销较小。我们将在附录 A 中讨论这种欺骗。

通过 `reduce(..)` 实现 `map(..)`，并不是表面上的明显的步骤，甚至是一种改善。然而，这种能力对于理解更高级的技术是至关重要的，如在附录 A 中的“转换”。

Filter 也是 Reduce

就像通过 `reduce(..)` 实现 `map(..)` 一样，也可以使用它实现 `filter(..)`：

```
var isOdd = v => v % 2 == 1;

[1,2,3,4,5].filter( isOdd );
// [1,3,5]

[1,2,3,4,5].reduce(
    (list,v) => (
        isOdd( v ) ? list.push( v ) : undefined,
        list
    ), []
);
// [1,3,5]
```

注意：这里有更加不纯的缩减器欺骗。不采用 `list.push(..)`，我们也可以采用 `list.concat(..)` 并返回合并后的新列表。我们将在附录 A 中继续介绍这个欺骗。

高级列表操作

现在，我们对这些基础的列表操作 `map(..)`、`filter(..)` 和 `reduce(..)` 感到比较舒服。让我们看看一些更复杂的操作，这些操作在某些场合下很有用。这些常用的实用函数存在于许多函数式编程的类库中。

去重

筛选列表中的元素，仅仅保留唯一的值。基于 `indexOf(..)` 函数查找（它采用 `===` 严格等于表达式）：

```
var unique =
    arr =>
        arr.filter(
            (v,idx) =>
                arr.indexOf( v ) == idx
        );
```

实现的原理是，当从左往右筛选元素时，列表项的 `idx` 位置和 `indexOf(..)` 找到的位置相等时，表明该列表项第一次出现，在这种情况下，将列表项加入到新数组中。

另一种实现 `unique(..)` 的方式是遍历 `arr`，当列表项不能在新列表中找到时，将其插入到新的列表中。这样可以采用 `reduce(..)` 来实现：

```
var unique =
    arr =>
        arr.reduce(
            (list,v) =>
                list.indexOf( v ) == -1 ?
                    ( list.push( v ), list ) : list
        , [] );
```

注意： 这里还有很多其他的方式实现这个去重算法，比如循环，并且其中不少还更高效，实现方式更聪明。然而，这两种方式的优点是，它们使用了内建的列表操作，它们能更方便的和其他列表操作链式／组合调用。我们会在本章的后面进一步讨论这些。

`unique(..)` 令人满意地产生去重后的新列表：

```
unique( [1,4,7,1,3,1,7,9,2,6,4,0,5,3] );
// [1, 4, 7, 3, 9, 2, 6, 0, 5]
```

扁平化

大多数时候，你看到的数组的列表项不是扁平的，很多时候，数组嵌套了数组，例如：

```
[ [1, 2, 3], 4, 5, [6, [7, 8]] ]
```

如果你想将其转化成下面的形式:

```
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
```

我们寻找的这个操作通常称为 `flatten(..)`。它可以采用如同瑞士军刀般的 `reduce(..)` 实现：

```
var flatten =
    arr =>
        arr.reduce(
            (list,v) =>
                list.concat( Array.isArray( v ) ? flatten( v ) : v )
        , [] );
```

注意： 这种处理嵌套列表的实现方式依赖于递归，我们将在后面的章节中进一步讨论。

在嵌套数组（任意嵌套层次）中使用 `flatten(..)`：

```
flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]] );
// [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
```

也许你会限制递归的层次到指定的层次。我们可以通过增加额外的 `depth` 形参来实现：

```
var flatten =
    (arr,depth = Infinity) =>
        arr.reduce(
            (list,v) =>
                list.concat(
                    depth > 0 ?
                        (depth > 1 && Array.isArray( v ) ?
                            flatten( v, depth - 1 ) :
                            v
                        ) :
                        [v]
                )
        , [] );
```

不同层级扁平化的结果如下所示：

```
flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 0 );
// [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]]

flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 1 );
// [0,1,2,3,4,[5,6,7],[8,[9,[10,[11,12],13]]]]

flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 2 );
// [0,1,2,3,4,5,6,7,8,[9,[10,[11,12],13]]]

flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 3 );
// [0,1,2,3,4,5,6,7,8,9,[10,[11,12],13]]

flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 4 );
// [0,1,2,3,4,5,6,7,8,9,10,[11,12],13]

flatten( [[0,1],2,3,[4,[5,6,7],[8,[9,[10,[11,12],13]]]]], 5 );
// [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
```

映射，然后扁平化

`flatten(..)` 的常用用法之一是当你映射一组元素列表，并且将每一项值从原来的值转换为数组。例如：

```
var firstNames = [
    { name: "Jonathan", variations: [ "John", "Jon", "Jonny" ] },
    { name: "Stephanie", variations: [ "Steph", "Stephy" ] },
    { name: "Frederick", variations: [ "Fred", "Freddy" ] }
];

firstNames
.map( entry => [entry.name].concat( entry.variations ) );
// [ ["Jonathan","John","Jon","Jonny"], ["Stephanie","Steph","Stephy"],
//   ["Frederick","Fred","Freddy"] ]
```

返回的值是二维数组，这样也许给处理带来一些不便。如果我们想得到所有名字的一维数组，我们可以对这个结果执行 `flatten(..)`：

```
flatten(
    firstNames
    .map( entry => [entry.name].concat( entry.variations ) )
);
// ["Jonathan","John","Jon","Jonny","Stephanie","Steph","Stephy","Frederick",
//  "Fred","Freddy"]
```

除了稍显啰嗦之外，将 `map(..)` 和 `flatten(..)` 采用独立的步骤的最主要的缺陷是关于性能方面。它会处理列表两次。

函数式编程的类库中，通常会定义一个 `flatMap(..)`（通常命名为 `chain(..)`）函数。这个函数将映射和之后的扁平化的操作组合起来。为了连贯性和组合（通过闭包）的简易性，`flatMap(..)` / `chain(..)` 实用函数的形参 `mapperFn, arr` 顺序通常和我们之前看到的独立的 `map(..)`、`filter(..)` 和 `reduce(..)` 一致。

```
flatMap( entry => [entry.name].concat( entry.variations ), firstNames );
// ["Jonathan","John","Jon","Jonny","Stephanie","Steph","Stephy","Frederick",
//  "Fred","Freddy"]
```

幼稚的采用独立的两步来实现 `flatMap(..)`：

```
var flatMap =
    (mapperFn,arr) =>
        flatten( arr.map( mapperFn ), 1 );
```

注意：我们将扁平化的层级指定为 `1`，因为通常 `flatMap(..)` 的定义是扁平化第一级。

尽管这种实现方式依旧会处理列表两次，带来了不好的性能。但我们可以将这些操作采用 `reduce(..)` 手动合并：

```
var flatMap =
    (mapperFn,arr) =>
        arr.reduce(
            (list,v) =>
                list.concat( mapperFn( v ) )
        , [] );
```

现在 `flatMap(..)` 方法带来了便利性和性能。有时你可能需要其他操作，比如和 `filter(..)` 混合使用。这样的话，将 `map(..)` 和 `flatten(..)` 独立开来始终更加合适。

Zip

到目前为止，我们介绍的列表操作都是操作单个列表。但是在某些情况下，需要操作多个列表。有一个闻名的操作：交替选择两个输入的列表中的值，并将得到的值组成子列表。这个操作被称之为 `zip(..)`：

```
zip( [1,3,5,7,9], [2,4,6,8,10] );
// [ [1,2], [3,4], [5,6], [7,8], [9,10] ]
```

选择值 `1` 和 `2` 到子列表 `[1,2]`，然后选择 `3` 和 `4` 到子列表 `[3,4]`，然后逐一选择。`zip(..)` 被定义为将两个列表中的值挑选出来。如果两个列表的的元素的个数不一致，这个选择会持续到较短的数组末尾时结束，另一个数组中多余的元素会被忽略。

一种 `zip(..)` 的实现：

```
function zip(arr1,arr2) {
    var zipped = [];
    arr1 = arr1.slice();
    arr2 = arr2.slice();

    while (arr1.length > 0 && arr2.length > 0) {
        zipped.push( [ arr1.shift(), arr2.shift() ] );
    }

    return zipped;
}
```

采用 `arr1.slice()` 和 `arr2.slice()` 可以确保 `zip(..)` 是纯的，不会因为接受到到数组引用造成副作用。

注意： 这个实现明显存在一些非函数式编程的思想。这里有一个命令式的 `while` 循环并且采用 `shift()` 和 `push(..)` 改变列表。在本书前面，我认为在纯函数中使用非纯的行为（通常是为了性能）是有道理的，只要其产生的副作用完全包含在这个函数内部。这种实现是安全纯净的。

合并

采用插入每个列表中的值的方式合并两个列表，如下所示：

```
mergeLists( [1,3,5,7,9], [2,4,6,8,10] );
// [1,2,3,4,5,6,7,8,9,10]
```

它可能不是那么明显，但其结果看上去和采用 `flatten(..)` 和 `zip(..)` 组合相似，代码如下:

```
zip( [1,3,5,7,9], [2,4,6,8,10] );
// [ [1,2], [3,4], [5,6], [7,8], [9,10] ]

flatten( [ [1,2], [3,4], [5,6], [7,8], [9,10] ] );
// [1,2,3,4,5,6,7,8,9,10]

// 组合后：
flatten( zip( [1,3,5,7,9], [2,4,6,8,10] ) );
// [1,2,3,4,5,6,7,8,9,10]
```

回顾 `zip(..)` ，他选择较短列表的最后一个值，忽视掉剩余的值； 而合并两个数组会很自然地保留这些额外的列表值。并且 `flatten(..)` 采用递归处理嵌套列表，但你可能只期望较浅地合并列表，保留嵌套的子列表。

这样，让我们定义一个更符合我们期望的 `mergeLists(..)` ：

```
function mergeLists(arr1,arr2) {
    var merged = [];
    arr1 = arr1.slice();
    arr2 = arr2.slice();

    while (arr1.length > 0 || arr2.length > 0) {
        if (arr1.length > 0) {
            merged.push( arr1.shift() );
        }
        if (arr2.length > 0) {
            merged.push( arr2.shift() );
        }
    }

    return merged;
}
```

注意： 许多函数式编程类库并不会定义 `mergeLists(..)` ，反而会定义 `merge(..)` 方法来合并两个对象的属性。这种 `merge(..)` 返回的结果和我们的 `mergeLists(..)` 不同。

另外，这里有一些选择采用缩减器实现合并列表的方法：

```
// 来自 @rwaldron
var mergeReducer =
    (merged,v,idx) =>
        (merged.splice( idx * 2, 0, v ), merged);


// 来自 @WebReflection
var mergeReducer =
    (merged,v,idx) =>
        merged
            .slice( 0, idx * 2 )
            .concat( v, merged.slice( idx * 2 ) );
```

采用 `mergeReducer(..)`：

```
[1,3,5,7,9]
.reduce( mergeReducer, [2,4,6,8,10] );
// [1,2,3,4,5,6,7,8,9,10]
```

提示：我们将在本章后面使用 `mergeReducer(..)` 这个技巧。

方法 vs 独立

对于函数式编程者来说，普遍感到失望的原因是 Javascript 采用统一的策略处理实用函数，但其中的一些也被作为独立函数提供了出来。想想在前面的章节中的介绍的大量的函数式编程实用程序，以及另一些实用函数是数组的原型方法，就像在这章中看到的那些。

当你想合并多个操作的时候，这个问题的痛苦程度更加明显：

```
[1,2,3,4,5]
.filter( isOdd )
.map( double )
.reduce( sum, 0 );                  // 18

//  采用独立的方法.

reduce(
    map(
        filter( [1,2,3,4,5], isOdd ),
        double
    ),
    sum,
    0
);                                  // 18
```

两种方式的 API 实现了同样的功能。但它们的风格完全不同。很多函数式编程者更倾向采用后面的方式，但是前者在 Javascript 中毫无疑问的更常见。后者特别地让人不待见之处是采用嵌套调用。人们更偏爱链式调用 —— 通常称为流畅的API风格，这种风格被 jQuery 和一些工具采用 —— 这种风格紧凑／简洁，并且可以采用声明式的自上而下的顺序阅读。

这种独立风格的手动合并的视觉顺序既不是严格的从左到右（自上而下），也不是严格的从右到左，而是从里往外。

从右往左（自下而上）这两种风格自动组成规范的阅读顺序。因此为了探索这些风格隐藏的差异，让我们特别的检查组合。他看上去应当简洁，但这两种情况都有点尴尬。

链式组合方法

这些数组方法接收绝对的 `this` 形参，因此尽管从外表上看，它们不能被当作一元运算看待，这会使组合更加尴尬。为了应对这些，我首先需要一个 `partial(..)` 版本的 `this`：

```
var partialThis =
    (fn,...presetArgs) =>
        // 故意采用 function 来为了 this 绑定
        function partiallyApplied(...laterArgs){
            return fn.apply( this, [...presetArgs, ...laterArgs] );
        };
```

我们也需要一个特殊的 `compose(..)`，它在上下文链中调用每一个部分应用的方法。它的输入值（即绝对的 this）由前一步传入：

```
var composeChainedMethods =
    (...fns) =>
        result =>
            fns.reduceRight(
                (result,fn) =>
                    fn.call( result )
                , result
            );
```

一起使用这两个 `this` 实用函数：

```
composeChainedMethods(
   partialThis( Array.prototype.reduce, sum, 0 ),
   partialThis( Array.prototype.map, double ),
   partialThis( Array.prototype.filter, isOdd )
)
( [1,2,3,4,5] );                    // 18
```

注意：那三个 `Array.prototype.XXX` 采用了内置的 `Array.prototype.*` 方法，这样我们可以在数组中重复使用它们。

独立组合实用函数

独立的 `compose(..)`，组合这些功能函数的风格不需要所有的这些广泛令人喜欢的 `this` 参数。例如，我们可以独立的定义成这样：

```
var filter = (arr,predicateFn) => arr.filter( predicateFn );

var map = (arr,mapperFn) => arr.map( mapperFn );

var reduce = (arr,reducerFn,initialValue) =>
    arr.reduce( reducerFn, initialValue );
```

但是，这种特别的独立风格给自身带来了不便。层级的数组上下文是第一个形参，而不是最后一个。因此我们需要采用右偏应用（right-partial application）来组合它们。

```
compose(
    partialRight( reduce, sum, 0 ),
    partialRight( map, double ),
    partialRight( filter, isOdd )
)
( [1,2,3,4,5] );                    // 18
```

这就是为何函数式编程类库通常定义 `filter(..)`、`map(..)` 和 `reduce(..)` 交替采用最后一个形参接收数组，而不是第一个。它们通常自动地柯理化实用函数：

```
var filter = curry(
    (predicateFn,arr) =>
        arr.filter( predicateFn )
);

var map = curry(
    (mapperFn,arr) =>
        arr.map( mapperFn )
);

var reduce = curry(
    (reducerFn,initialValue,arr) =>
        arr.reduce( reducerFn, initialValue );
```

采用这种方式定义实用函数，组合流程会显得更加友好：

```
compose(
    reduce( sum )( 0 ),
    map( double ),
    filter( isOdd )
)
( [1,2,3,4,5] );                    // 18
```

这种很整洁的实现方式，就是函数式编程者喜欢独立的实用程序风格，而不是实例方法的原因。但这种情况因人而异。

方法适配独立

在前面的 `filter(..)` / `map(..)` / `reduce(..)` 的定义中，你可能发现了这三个方法的共同点：它们都派发到相对应的原生数组方法。因此，我们能采用实用函数生成这些独立适配函数吗？当然可以，让我们定义 `unboundMethod(..)` 来做这些：

```
var unboundMethod =
    (methodName,argCount = 2) =>
        curry(
            (...args) => {
                var obj = args.pop();
                return obj[methodName]( ...args );
            },
            argCount
        );
```

使用这个实用函数：

```
var filter = unboundMethod( "filter", 2 );
var map = unboundMethod( "map", 2 );
var reduce = unboundMethod( "reduce", 3 );

compose(
    reduce( sum )( 0 ),
    map( double ),
    filter( isOdd )
)
( [1,2,3,4,5] );                    // 18
```

注意: `unboundMethod(..)` 在 Ramda 中称之为 `invoker(..)` 。

独立函数适配为方法

如果你喜欢仅仅使用数组方法（流畅的链式风格），你有两个选择：

1. 采用额外的方法扩展内建的 `Array.prototype`
2. 把独立实用函数适配成一个缩减函数，并且将其传递给 `reduce(..)` 实例方法。

不要采用第一种 扩展诸如 `Array.prototype` 的原生方法从来不是一个好主意，除非定义一个 `Array` 的子类。但是这超出了这里的讨论范围。为了不鼓励这种不好的习惯，我们不会进一步去探讨这种方式。

让我们关注第二种。为了说明这点，我们将前面定义的递归实现的 `flatten(..)` 转换为独立实用函数：

```
var flatten =
    arr =>
        arr.reduce(
            (list,v) =>
                list.concat( Array.isArray( v ) ? flatten( v ) : v )
        , [] );
```

让我们将里面的 `reducer(..)` 函数抽取成独立的实用函数（并且调整它，让其独立于外部的 `flatten(..)` 运行）：

```
// 刻意使用具名函数用于递归中的调用
function flattenReducer(list,v) {
    return list.concat(
        Array.isArray( v ) ? v.reduce( flattenReducer, [] ) : v
    );
}
```

现在，我们可以在数组方法链中通过 `reduce(..)` 调用这个实用函数：

```
[ [1, 2, 3], 4, 5, [6, [7, 8]] ]
.reduce( flattenReducer, [] )
// ..
```

查寻列表

到此为止，大部分示例有点无聊，它们基于一列数字或者字符串，让我们讨论一些有亮点的列表操作：声明式地建模一些命令式语句。

看看这个基本例子：

```
var getSessionId = partial( prop, "sessId" );
var getUserId = partial( prop, "uId" );

var session, sessionId, user, userId, orders;

session = getCurrentSession();
if (session != null) sessionId = getSessionId( session );
if (sessionId != null) user = lookupUser( sessionId );
if (user != null) userId = getUserId( user );
if (userId != null) orders = lookupOrders( userId );
if (orders != null) processOrders( orders );
```

首先，我们可以注意到声明和运行前的一系列 If 语句确保了由 `getCurrentSession()` 、 `getSessionId(..)` 、 `lookupUser(..)` 、 `getUserId(..)` 、 `lookupOrders(..)` 和 `processOrders(..)` 这六个函数组合调用时的有效。理想地，我们期望摆脱这些变量定义和命令式的条件。

不幸的是，在第 4 章中讨论的 `compose(..)` ／ `pipe(..)` 实用函数并没有提供给一个便捷的方式来表达在这个组合中的 `!= null` 条件。让我们定义一个实用函数来解决这个问题：

```
var guard =
    fn =>
        arg =>
            arg != null ? fn( arg ) : arg;
```

这个 `guard(..)` 实用函数让我们映射这五个条件确保函数：

```
[ getSessionId, lookupUser, getUserId, lookupOrders, processOrders ]
.map( guard )
```

这个映射的结果是组合的函数数组（事实上，这是个有列表顺序的管道）。我们可以展开这个数组到 `pipe(..)` ，但由于我们已经做列表操作，让我们采用 `reduce(..)` 来处理。采用 `getCurrentSession()` 返回的会话值作为初始值：

```
.reduce(
    (result,nextFn) => nextFn( result )
    , getCurrentSession()
)
```

接下来，我们观察到 `getSessionId(..)` 和 `getUserId(..)` 可以看成对应的 `"sessId"` 和 `"uId"` 的映射：

```
[ "sessId", "uId" ].map( propName => partial( prop, propName ) )
```

但是为了使用这些，我们需要将另外三个函数（`lookupUser(..)`、`lookupOrders(..)` 和 `processOrders(..)`）插入进来，用来获取上面讨论的那五个守护／组合函数。

为了实现插入，我们采用列表合并来模拟这些。回顾本章前面介绍的 `mergeReducer(..)`：

```
var mergeReducer =
    (merged,v,idx) =>
        (merged.splice( idx * 2, 0, v ), merged);
```

我们可以采用 `reduce(..)`（我们的瑞士军刀，还记得吗？）在生成的 `getSessionId(..)` 和 `getUserId(..)` 函数之间的数组中“插入” `lookupUser(..)`，通过合并这两个列表：

```
.reduce( mergeReducer, [ lookupUser ] )
```

然后我们将 `lookupOrders(..)` 和 `processOrders(..)` 加入到正在执行的函数数组末尾：

```
.concat( lookupOrders, processOrders )
```

总结下，生成的五个函数组成的列表表达为：

```
[ "sessId", "uId" ].map( propName => partial( prop, propName ) )
.reduce( mergeReducer, [ lookupUser ] )
.concat( lookupOrders, processOrders )
```

最后，将所有函数合并到一起，将这些函数数组添加到之前的守护和组合上：

```
[ "sessId", "uId" ].map( propName => partial( prop, propName ) )
.reduce( mergeReducer, [ lookupUser ] )
.concat( lookupOrders, processOrders )
.map( guard )
.reduce(
    (result,nextFn) => nextFn( result )
    , getCurrentSession()
);
```

所有必要的变量声明和条件一去不复返了，取而代之的是采用整洁和声明式的列表操作链接在一起。

如果你觉得现在的这个版本比之前要难，不要担心。毫无疑问的，前面的命令式的形式，你可能更加熟悉。进化为函数式编程者的一步就是开发一些具有函数式编程风格的代码，比如这些列表操作。随着时间推移，我们跳出这些代码，当你切换到声明式风格时更容易感受到代码的可读性。

在离开这个话题之前，让我们做一个真实的检查：这里的示例过于造作。不是所有的代码片段被简单的采用列表操作模拟。务实的获取方式是本能的寻找这些机会，而不是过于追求代码的技巧；一些改进比没有强。经常退一步，并且问自己，是提升了还是损害了代码的可读性。

融合

当你更多的考虑在代码中使用函数式列表操作，你可能会很快地开始看到链式组合行为，如：

```
..
.filter(..)
.map(..)
.reduce(..);
```

往往，你可能会把多个相邻的操作用链式来调用，比如：

```
someList
.filter(..)
.filter(..)
.map(..)
.map(..)
.map(..)
.reduce(..);
```

好消息是，链式风格是声明式的，并且很容易看出详尽的执行步骤和顺序。它的不足之处在于每一个列表操作都需要循环整个列表，意味着不必要的性能损失，特别是在列表非常长的时候。

采用交替独立的风格，你可能看到的代码如下：

```
map(
    fn3,
    map(
        fn2,
        map( fn1, someList )
    )
);
```

采用这种风格，这些操作自下而上列出，这依然会循环数组三遍。

融合处理了合并相邻的操作，这样可以减少列表的迭代次数。这里我们关注于合并相邻的 `map(..)`，这很容易解释。

想象一下这样的场景：

```
var removeInvalidChars = str => str.replace( /[^\w]*/g, "" );

var upper = str => str.toUpperCase();

var elide = str =>
    str.length > 10 ?
        str.substr( 0, 7 ) + "..." :
        str;

var words = "Mr. Jones isn't responsible for this disaster!"
    .split( /\s/ );

words;
// ["Mr.","Jones","isn't","responsible","for","this","disaster!"]

words
.map( removeInvalidChars )
.map( upper )
.map( elide );
// ["MR","JONES","ISNT","RESPONS...","FOR","THIS","DISASTER"]
```

注意在这个转换流程中的每一个值。在 `words` 列表中的第一个值，开始为 `"Mr."`，变为 `"Mr"`，然后为 `"MR"`，然后通过 `elide(..)` 不变。另一个数据流为：`"responsible"` -> `"responsible"` -> `"RESPONSIBLE"` -> `"RESPONS..."`。

换句话说，你可以将这些数据转换看成这样：

```
elide( upper( removeInvalidChars( "Mr." ) ) );
// "MR"

elide( upper( removeInvalidChars( "responsible" ) ) );
// "RESPONS..."
```

你抓住重点了吗？我们可以将那三个独立的相邻的 `map(..)` 调用步骤看成一个转换组合。因为它们都是一元函数，并且每一个返回值都是下一个点输入值。我们可以采用 `compose(..)` 执行映射功能，并将这个组合函数传入到单个 `map(..)` 中调用：

```
words
.map(
    compose( elide, upper, removeInvalidChars )
);
// ["MR","JONES","ISNT","RESPONS...","FOR","THIS","DISASTER"]
```

这是另一个 `pipe(..)` 能更便利的方式处理组合的场景，这样可读性很有条理：

```
words
.map(
    pipe( removeInvalidChars, upper, elide )
);
// ["MR","JONES","ISNT","RESPONS...","FOR","THIS","DISASTER"]
```

如何融合两个以上的 `filter(..)` 谓词函数呢？通常视为一元函数，它们似乎适合组合。但是有个小问题，每一个函数返回了不同类型的值（ `boolean` ），这些返回值并不是下一个函数需要的输入参数。融合相邻的 `reduce(..)` 调用也是可能的，但缩减器并不是一元的，这也会带来不小的挑战。我们需要更复杂的技巧来实现这些融合。我们将在附录 A 的“转换”中讨论这些高级方法。

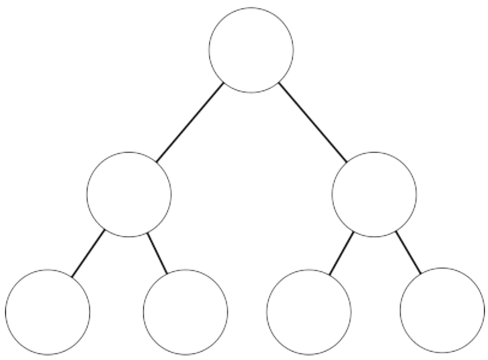
列表之外

到目前为止，我们讨论的操作都是在列表（数组）数据结构中，这是迄今为止你遇到的最常见的场景。但是更普遍的意义是，这些操作可以在任一集合执行。

就像我们之前说过，数组的 `map(..)` 方法对数组中的每一个值做单值操作，任何数据结构都可以采用 `map(..)` 操作做类似的事情。同样的，也可以实现 `filter(..)` ， `reduce(..)` 和其他能工作于这些数据结构的值的操作。

函数式编程精神中重要的部分是这些操作必须依赖值的不变性，意味着它们必须返回一个新的值，而不是改变存在的值。

让我们描述那个广为人知的数据结构：二叉树。二叉树指的是一个节点（只有一个对象！）有两个字节点（这些字节点也是二叉树），这两个字节点通常称之为左和右子树。树中的每个节点包含总体数据结构的值。



在这个插图中，我们将我们的二叉树描述为二叉搜索树（BST）。然而，树的操作和其他非二叉搜索树没有区别。

注意： 二叉搜索树是特定的二叉树，该树中的节点值彼此之间存在特定的约束关系。每个树中的左子节点的值小于根节点的值，跟子节点的值也小于右子节点的值。这里“小于”的概念是相对于树中存储数据的类型。它可以是数字的数值，也可以是字符串在词典中的顺序，等

等。二叉搜索树的价值在于在处理在树中搜索一个值非常高效便捷，采用一个递归的二叉搜索算法。

让我们采用这个工厂函数创建二叉树对象：

```
var BinaryTree =
    (value,parent,left,right) => ({ value, parent, left, right });
```

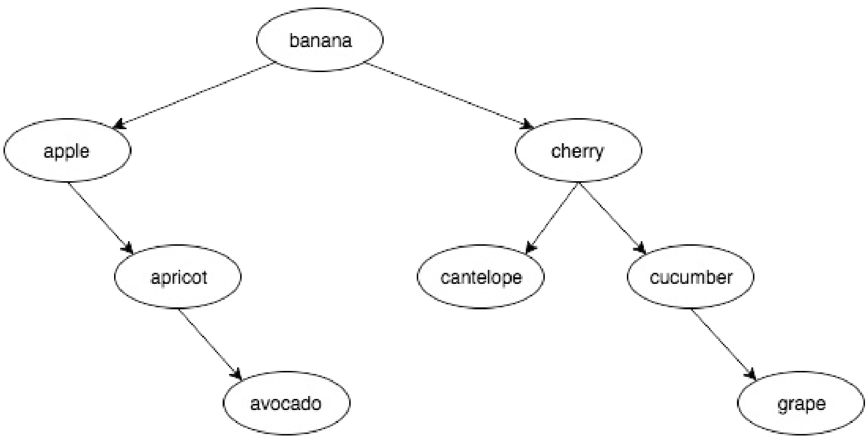
为了方便，我们在每个Node中不仅仅保存了 `left` 和 `right` 子树节点，也保存了其自身的 `parent` 节点引用。

现在，我们将一些常见的产品名（水果，蔬菜）定义为二叉搜索树：

```
var banana = BinaryTree( "banana" );
var apple = banana.left = BinaryTree( "apple", banana );
var cherry = banana.right = BinaryTree( "cherry", banana );
var apricot = apple.right = BinaryTree( "apricot", apple );
var avocado = apricot.right = BinaryTree( "avocado", apricot );
var cantelope = cherry.left = BinaryTree( "cantelope", cherry );
var cucumber = cherry.right = BinaryTree( "cucumber", cherry );
var grape = cucumber.right = BinaryTree( "grape", cucumber );
```

在这个树形结构中，`banana` 是根节点，这棵树可能采用不同的方式创建节点，但其依旧可以采用二叉搜索树一样的方式访问。

这棵树如下图所示：



这里有多种方式来遍历一颗二叉树来处理它的值。如果这棵树是二叉搜索树，我们还可以有序的遍历它。通过先访问左侧子节点，然后自身节点，最后右侧子节点，这样我们可以得到升序排列的值。

现在，你不能仅仅通过像在数组中用 `console.log(..)` 打印出二叉树。我们先定义一个便利的方法，主要用来打印。定义的 `forEach(..)` 方法能像和数组一样的方式来访问二叉树：

```
// 顺序遍历
BinaryTree.forEach = function forEach(visitFn,node){
    if (node) {
        if (node.left) {
            forEach( visitFn, node.left );
        }

        visitFn( node );

        if (node.right) {
            forEach( visitFn, node.right );
        }
    }
};
```

注意： 采用递归处理二叉树更自然。我们的 `forEach(..)` 实用函数采用递归调用自身来处理左右字节点。我们将在后续的章节章深入讨论递归。

回顾在本章开头描述的 `forEach(..)`，它存在有用的副作用，通常函数式编程期望有这个副作用。在这种情况下，我们仅仅在 I／O 的副作用下使用 `forEach(..)`，因此它是完美的理想的辅助函数。

采用 `forEach(..)` 打印那个二叉树中的值：

```
BinaryTree.forEach( node => console.log( node.value ), banana );
// apple apricot avocado banana cantelope cherry cucumber grape

// 仅访问根节点为 `cherry` 的子树
BinaryTree.forEach( node => console.log( node.value ), cherry );
// cantelope cherry cucumber grape
```

为了采用函数式编程的方式操作我们定义的那个二叉树，我们定义一个 `map(..)` 函数：

```
BinaryTree.map = function map(mapperFn,node){
    if (node) {
        let newNode = mapperFn( node );
        newNode.parent = node.parent;
        newNode.left = node.left ?
            map( mapperFn, node.left ) : undefined;
        newNode.right = node.right ?
            map( mapperFn, node.right ): undefined;

        if (newNode.left) {
            newNode.left.parent = newNode;
        }
        if (newNode.right) {
            newNode.right.parent = newNode;
        }

        return newNode;
    }
};
```

你可能会认为采用 `map(..)` 仅仅处理节点的 `value` 属性，但通常情况下，我们可能需要映射树的节点本身。因此，`mapperFn(..)` 传入整个访问的节点，在应用了转换之后，它期待返回一个全新的 `BinaryTree(..)` 节点回来。如果你返回了同样的节点，这个操作会改变你的树，并且很可能会引起意想不到的结果！

让我们映射我们的那个树，得到一列大写产品名：

```
var BANANA = BinaryTree.map(
    node => BinaryTree( node.value.toUpperCase() ),
    banana
);

BinaryTree.forEach( node => console.log( node.value ), BANANA );
// APPLE APRICOT AVOCADO BANANA CANTELOPE CHERRY CUCUMBER GRAPE
```

`BANANA` 和 `banana` 是一个不同的树（所有的节点都不同），就像在列表中执行 `map(..)` 返回一个新的数组。就像其他对象／数组的数组，如果 `node.value` 本身是某个对象／数组的引用，如果你想做深层次的转换，那么你就需要在映射函数中手动的对它做深拷贝。

如何处理 `reduce(..)`？相同的基本处理过程：有序遍历树的节点的方式。一种可能的用法是 `reduce(..)` 我们的树得到它的值的数组。这对将来适配其他典型的列表操作很有帮助。或者，我们可以 `reduce(..)` 我们的树，得到一个合并了它所有产品名的字符串。

我们模仿数组中 `reduce(..)` 的行为，它接受那个可选的 `initialValue` 参数。该算法有一点难度，但依旧可控：

```
BinaryTree.reduce = function reduce(reducerFn,initialValue,node){
    if (arguments.length < 3) {
        // 移动参数，直到 `initialValue` 被删除
        node = initialValue;
    }

    if (node) {
        let result;

        if (arguments.length < 3) {
            if (node.left) {
                result = reduce( reducerFn, node.left );
            }
            else {
                return node.right ?
                    reduce( reducerFn, node, node.right ) :
                    node;
            }
        }
        else {
            result = node.left ?
                reduce( reducerFn, initialValue, node.left ) :
                initialValue;
        }

        result = reducerFn( result, node );
        result = node.right ?
            reduce( reducerFn, result, node.right ) : result;
        return result;
    }

    return initialValue;
};
```

让我们采用 `reduce(..)` 产生一个购物单（一个数组）：

```
BinaryTree.reduce(
    (result,node) => result.concat( node.value ),
    [],
    banana
);
// ["apple","apricot","avocado","banana","cantelope"
//   "cherry","cucumber","grape"]
```

最后，让我们考虑在树中用 `filter(..)` 。这个算法迄今为止最棘手，因为它有效（实际上没有）影响从树上删除节点，这需要处理几个问题。不要被这种实现吓到。如果你喜欢，现在跳过它，关注我们如何使用它而不是实现。

```
BinaryTree.filter = function filter(predicateFn,node){
```

```
    if (node) {
        let newNode;
        let newLeft = node.left ?
            filter( predicateFn, node.left ) : undefined;
        let newRight = node.right ?
            filter( predicateFn, node.right ) : undefined;

        if (predicateFn( node )) {
            newNode = BinaryTree(
                node.value,
                node.parent,
                newLeft,
                newRight
            );
            if (newLeft) {
                newLeft.parent = newNode;
            }
            if (newRight) {
                newRight.parent = newNode;
            }
        }
        else {
            if (newLeft) {
                if (newRight) {
                    newNode = BinaryTree(
                        undefined,
                        node.parent,
                        newLeft,
                        newRight
                    );
                    newLeft.parent = newRight.parent = newNode;

                    if (newRight.left) {
                        let minRightNode = newRight;
                        while (minRightNode.left) {
                            minRightNode = minRightNode.left;
                        }

                        newNode.value = minRightNode.value;

                        if (minRightNode.right) {
                            minRightNode.parent.left =
                                minRightNode.right;
                            minRightNode.right.parent =
                                minRightNode.parent;
                        }
                        else {
                            minRightNode.parent.left = undefined;
                        }

                        minRightNode.right =
                            minRightNode.parent = undefined;
                    }
```

```
                    else {
                        newNode.value = newRight.value;
                        newNode.right = newRight.right;
                        if (newRight.right) {
                            newRight.right.parent = newNode;
                        }
                    }
                }
                else {
                    return newLeft;
                }
            }
            else {
                return newRight;
            }
        }

        return newNode;
    }
};
```

这段代码的大部分是为了专门处理当存在重复的树形结构中的节点被“删除”（过滤掉）的时候，移动节点的父／子引用。

作为一个描述使用 `filter(..)` 的例子，让我们产生仅仅包含蔬菜的树：

```
var vegetables = [ "asparagus", "avocado", "brocolli", "carrot",
    "celery", "corn", "cucumber", "lettuce", "potato", "squash",
    "zucchini" ];

var whatToBuy = BinaryTree.filter(
    // 将蔬菜从农产品清单中过滤出来
    node => vegetables.indexOf( node.value ) != -1,
    banana
);

// 购物清单
BinaryTree.reduce(
    (result,node) => result.concat( node.value ),
    [],
    whatToBuy
);
// ["avocado","cucumber"]
```

你会在简单列表中使用本章大多数的列表操作。但现在你发现这个概念适用于你可能需要的任何数据结构和操作。函数式编程可以广泛应用在许多不同的场景，这是非常强大的！

总结

三个强大通用的列表操作：

- `map(..)`：转换列表项的值到新列表。
- `filter(..)`：选择或过滤掉列表项的值到新数组。
- `reduce(..)`：合并列表中的值，并且产生一个其他的值（经常但不总是非列表的值）。

其他一些非常有用的处理列表的高级操作：`unique(..)`、`flatten(..)` 和 `merge(..)`。

融合采用函数组合技术来合并多个相邻的 `map(..)` 调用。这是常见的性能优化方式，并且它也使得列表操作更加自然。

列表通常以数组展现，但它也可以作为任何数据结构表达／产生一个有序的值集合。因此，所有这些“列表操作”都是“数据结构操作”。

Javascript 轻量级函数式编程

第 9 章：递归

在下一页，我们将进入到递归的论题。

(本页剩余部分故意留白)

我们来谈谈递归吧。在我们入坑之前，请查阅上一页的正式定义。

我知道，这个笑话弱爆了 :)

大部分的开发人员都承认递归是一门非常强大的编程技术，但他们并不喜欢去使用它。在这个意义上，我把它放在与正则表达式相同的类别中。递归技术强大但又令人困惑，因此被视为 不值得我们投入努力。

我是递归编程的超级粉丝，你，也可以的！在这一章节中我的目标就是说服你：递归是一个重要的工具，你应该将它用在你的函数式编程中。当你正确使用时，递归编程可以轻松地描述复杂问题。

定义

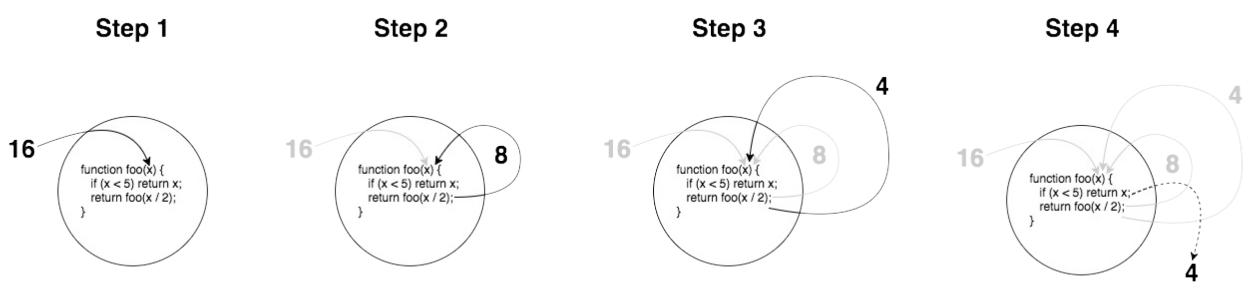
所谓递归，是当一个函数调用自身，并且该调用做了同样的事情，这个循环持续到基本条件满足时，调用循环返回。

警告： 如果你不能确保基本条件是递归的 终结者，递归将会一直执行下去，并且会把你的项目损坏或锁死；恰当的基本条件十分重要！

但是... 这个定义的书面形式太让人疑惑了。我们可以做的更好些。思考下这个递归函数：

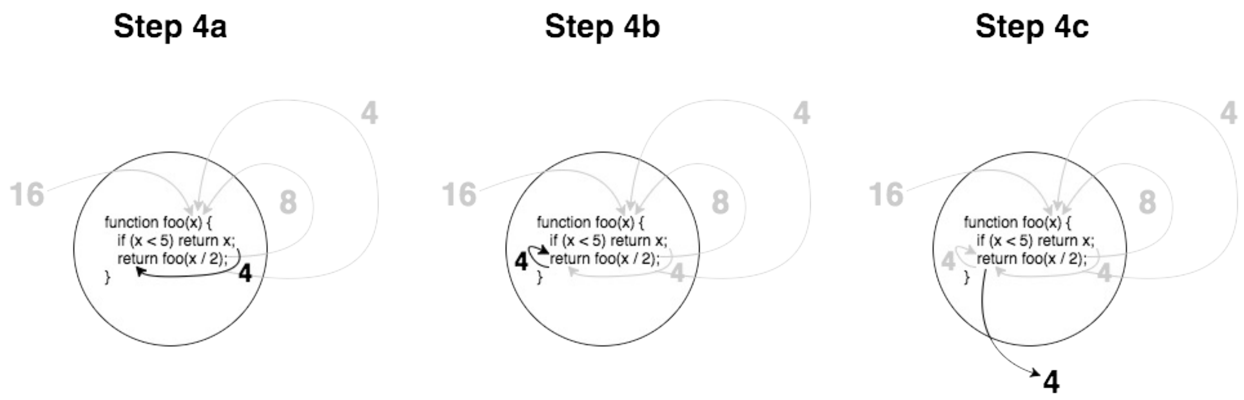
```
function foo(x) {
    if (x < 5) return x;
    return foo( x / 2 );
}
```

设想一下，如果我们调用 `foo(16)` 将会发生什么：



在 step 2 中，`x / 2` 的结果是 `8`， 这个结果以参数的形式传递到 `foo(..)` 并运行。同样的，在 step 3 中， `x / 2` 的结果是 `4`，这个结果以参数的形式传递到另一个 `foo(..)` 并运行。但愿我解释得足够直白。

但是一些人经常会在 step 4 中卡壳。一旦我们满足了基本条件 `x` (值为4) `< 5`，我们将不再调用递归函数，只是（有效地）执行了 `return 4`。 特别是图中返回 `4` 的虚线那块，它简化了那里的过程，因此我们来深入了解最后一步，并把它折分为三个子步骤：



该次的返回值会回过头来触发调用栈中所有的函数调用（并且它们都执行 `return` ）。

另外一个递归实例：

```
function isPrime(num,divisor = 2){
    if (num < 2 || (num > 2 && num % divisor == 0)) {
        return false;
    }
    if (divisor <= Math.sqrt( num )) {
        return isPrime( num, divisor + 1 );
    }

    return true;
}
```

这个质数的判断主要是通过验证，从2到 `num` 的平方根之间的每个整数，看是否存在某一整数可以整除 `num` (`%` 求余结果为 `0`)。如果存在这样的整数，那么 `num` 不是质数。反之，是质数。 `divisor + 1` 使用递归来遍历每个可能的 `divisor` 值。

递归的最著名的例子之一是计算斐波那契数，该数列定义如下：

```
fib( 0 ): 0
fib( 1 ): 1
fib( n ):
    fib( n - 2 ) + fib( n - 1 )
```

注意: 数列的前几个数值是： 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... 每一个数字都是数列中前两个数字之和。

直接用代码来定义斐波那契：

```
function fib(n) {
    if (n <= 1) return n;
    return fib( n - 2 ) + fib( n - 1 );
}
```

函数 `fib(..)` 对自身进行了两次递归调用，这通常叫作二分递归查找。后面我们将会更多地讨论二分递归查找。

在整个章节中，我们将会用不同形式的 `fib(..)` 来说明关于递归的想法，但不太好的地方就是，这种特殊的方式会造成很多重复性的工作。 `fib(n-1)` 和 `fib(n-2)` 运行时候两者之间并没有任何的共享，但做的事情几乎又完全相同，这种情况一直持续到整个整数空间(译者注：形参 `n`)降到 `0` 。

在第五章的性能优化方面我们简单的谈到了记忆存储技术。本章中，记忆存储技术使得任意一个传入到 `fib(..)` 的数值只会被计算一次而不是多次。虽然我们不会在这里过多地讨论这个技术话题，但不论是递归或其它任何算法，我们都要谨记，性能优化是非常重要的。

相互递归

当一个函数调用自身时，很明显，这叫作直接递归。比如前面部分我们谈到的 `foo(..)` ， `isPrime(..)` 以及 `fib(..)` 。如果在一个递归循环中，出现两个及以上的函数相互调用，则称之为相互递归。

这两个函数就是相互递归：

```
function isOdd(v) {
    if (v === 0) return false;
    return isEven( Math.abs( v ) - 1 );
}

function isEven(v) {
    if (v === 0) return true;
    return isOdd( Math.abs( v ) - 1 );
}
```

是的，这个奇偶数的判断笨笨的。但也给我们提供了一些思路：某些算法可以根据相互递归来定义。

回顾下上节中的二分递归法 `fib(..)` ；我们可以换成相互递归来表示：

```
function fib_(n) {
    if (n == 1) return 1;
    else return fib( n - 2 );
}

function fib(n) {
    if (n == 0) return 0;
    else return fib( n - 1 ) + fib_( n );
}
```

注意： `fib(..)` 相互递归的实现方式改编自“用相互递归来实现斐波纳契数列”研究报告(https://www.researchgate.net/publication/246180510_Fibonacci_Numbers_Using_Mutual_Recursion) 。

虽然这些相互递归的示例有点不切实际，但是在更复杂的使用场景下，相互递归是非常有用的。

为什么选择递归？

现在我们已经给出了递归的定义和说明，下面来看下，为什么说递归是有用的。

递归深谙函数式编程之精髓，最被广泛引证的原因是，在调用栈中，递归把(大部分)显式状态跟踪换为了隐式状态。通常，当问题需要条件分支和回溯计算时，递归非常有用，此外在纯迭代环境中管理这种状态，是相当棘手的；最起码，这些代码是不可或缺且晦涩难懂。但是在堆栈上调用每一级的分支作为其自己的作用域，很明显，这通常会影响到代码的可读性。

简单的迭代算法可以用递归来表达：

```
function sum(total,...nums) {
    for (let i = 0; i < nums.length; i++) {
        total = total + nums[i];
    }

    return total;
}

// vs

function sum(num1,...nums) {
    if (nums.length == 0) return num1;
    return num1 + sum( ...nums );
}
```

我们不仅用调用栈代替了 `for` 循环，而且用 `return` s 的形式在回调栈中隐式地跟踪增量的求和（ `total` 的间歇状态），而非在每次迭代中重新分配 `total` 。通常，FPer 倾向于尽可能地避免重新分配局部变量。

像我们总结的那样，在基本算法里，这些差异是微乎其微的。但是，随着算法复杂度的提升，你将更加能体会到递归带来的收益，而不是这些命令式状态跟踪。

声明式递归

数学家使用 **Σ** 符号来表示一列数字的总和。主要原因是，如果他们使用更复杂的公式而且不得不手动书写求和的话，会造成更多麻烦（而且会降低阅读性！），比如 `1 + 3 + 5 + 7 + 9 + ..` 。符号是数学的声明式语言！

正如 **Σ** 是为运算而声明，递归是为算法而声明。递归说明：一个问题存在解决方案，但并不一定要求阅读代码的人了解该解决方案的工作原理。我们来思考下找出入参最大偶数值的两种方法：

```
function maxEven(...nums) {
    var num = -Infinity;

    for (let i = 0; i < nums.length; i++) {
        if (nums[i] % 2 == 0 && nums[i] > num) {
            num = nums[i];
        }
    }

    if (num !== -Infinity) {
        return num;
    }
}
```

这种实现方式不是特别难处理，但它的一些细微的问题也不容忽视。很明显，运行 `maxEven()` ， `maxEven(1)` 和 `maxEven(1,13)` 都将会返回 `undefined` ？最终的 `if` 语句是必需的吗？

我们试着换一个递归的方法来对比下。我们用下面的符号来表示递归：

```
maxEven( nums ):
    maxEven( nums.0, maxEven( ...nums.1 ) )
```

换句话说，我们可以将数字列表的 max-even 定义为其余数字的 max-even 与第一个数字的 max-even 的结果。例如：

```
maxEven( 1, 10, 3, 2 ):
    maxEven( 1, maxEven( 10, maxEven( 3, maxEven( 2 ) ) )
```

在 JS 中实现这个递归定义的方法之一是：

```
function maxEven(num1,...restNums) {
    var maxRest = restNums.length > 0 ?
            maxEven( ...restNums ) :
            undefined;

    return (num1 % 2 != 0 || num1 < maxRest) ?
        maxRest :
        num1;
}
```

那么这个方法有什么优点吗？

首先，参数与之前不一样了。我专门把第一个参数叫作 `num1`，剩余的其它参数放在一起叫作 `restNums`。我们本可以把所有参数都放在 `nums` 数组中，并从 `nums[0]` 获取第一个参数。这是为什么呢？

函数的参数是专门为递归定义的。它看起来像这样：

```
maxEven( num1, ...restNums ):
    maxEven( num1, maxEven( ...restNums ) )
```

你有发现参数和递归之间的相似性吗？

当我们在函数体签名中进一步提升递归的定义，函数的声明也会得到提升。如果我们能够把递归的定义从参数反映到函数体中，那就更棒了。

但我想说最明显的改进是，`for` 循环造成的错乱感没有了。所有循环逻辑都被抽象为递归回调栈，所以这些东西不会造成代码混乱。我们可以轻松的把精力集中在一次比较两个数字来找到最大偶数值的逻辑中 —— 不管怎么说，这都是很重要的部分！

从思想上来讲，这如同一位数学家在更庞大的方程中使用 **Σ** 求和一样。我们说，“数列中剩余值的最大偶数是通过 `maxEven(...restNums)` 计算出来的，所以我们只需要继续推断这一部分。”

另外，我们用 `restNums.length > 0` 保证推断更加合理，因为当没有参数的情况下，返回的 `maxRest` 结果肯定是 `undefined`。我们不需要对这部分的推理投入额外的精力。这个基本条件（没有参数情况下）显而易见。

接下来，我们把精力放在对比 `num1` 和 `maxRest` 上 —— 算法的主要逻辑是如何确定两个数字中的哪一个（如果有的话）是最大偶数。如果 `num1` 不是偶数（`num1 % 2 != 0`），或着它小于 `maxRest`，那么，即使 `maxRest` 的值是 `undefined`，`maxRest` 会 `return` 掉。否则，返回结果会是 `num1`。

在阅读整个实现过程中，与命令式的方法相比，我所做这个例子的推理过程更加直接，核心点更加突出，少做无用功；比 `for` 循环中引用 `无穷数值` 这一方法 更具有声明性。

小贴士： 我们应该指出，除了手动迭代或递归之外，另一种（可能更好的）建模的方法是我们在在第7章中讨论的列表操作。我们先把数列中的偶数用 `filter(..)` 过滤出来，然后通过递归 `reduce(..)` 函数（对比两个数值并返回其中较大的数值）来找到最大值。在这里，我们只是使用这个例子来说明在手动迭代中递归的声明性更强。

还有一个递归的例子：计算二叉树的深度。二叉树的深度是指通过树的节点向下（左或右）的最长路径。还有另一种通过递归来定义的方式：任何树节点的深度为1（当前节点）加上来自其左侧或右侧子树的深度的最大值：

```
depth( node ):
    1 + max( depth( node.left ), depth( node.right ) )
```

直接转换为二分法递归函数：

```
function depth(node) {
    if (node) {
        let depthLeft = depth( node.left );
        let depthRight = depth( node.right );
        return 1 + max( depthLeft, depthRight );
    }

    return 0;
}
```

我不打算列出这个算法的命令式形式，但请相信我，它太麻烦、过于命令式了。这种递归方法很不错，声明也很优雅。它遵循递归的定义，与递归定义的算法非常接近，省心。

并不是所有的问题都是完全可递归的。它不是你可以广泛应用的灵丹妙药。但是递归可以非常有效地将问题的表达，从更具必要性转变为更有声明性。

栈、堆

一起看下之前的两个递归函数 `isOdd(..)` 和 `isEven(..)` ：

```
function isOdd(v) {
    if (v === 0) return false;
    return isEven( Math.abs( v ) - 1 );
}

function isEven(v) {
    if (v === 0) return true;
    return isOdd( Math.abs( v ) - 1 );
}
```

如果你执行下面这行代码，在大多数浏览器里面都会报错：

```
isOdd( 33333 );            // RangeError: Maximum call stack size exceeded
```

这个错误是什么情况？引擎抛出这个错误，是因为它试图保护系统内存不会被你的程序耗尽。为了解释这个问题，我们需要先看看当函数调用时JS引擎中发生了什么。

每个函数调用都将开辟出一小块称为堆栈帧的内存。堆栈帧中包含了函数语句当前状态的某些重要信息，包括任意变量的值。之所以这样，是因为一个函数暂停去执行另外一个函数，而另外一个函数运行结束后，引擎需要返回到之前暂停时候的状态继续执行。

当第二个函数开始执行，堆栈帧增加到 2 个。如果第二个函数又调用了另外一个函数，堆栈帧将增加到 3 个，以此类推。“栈”的意思是，函数被它前一个函数调用时，这个函数帧会被“推”到最顶部。当这个函数调用结束后，它的帧会从堆栈中退出。

看下这段程序：

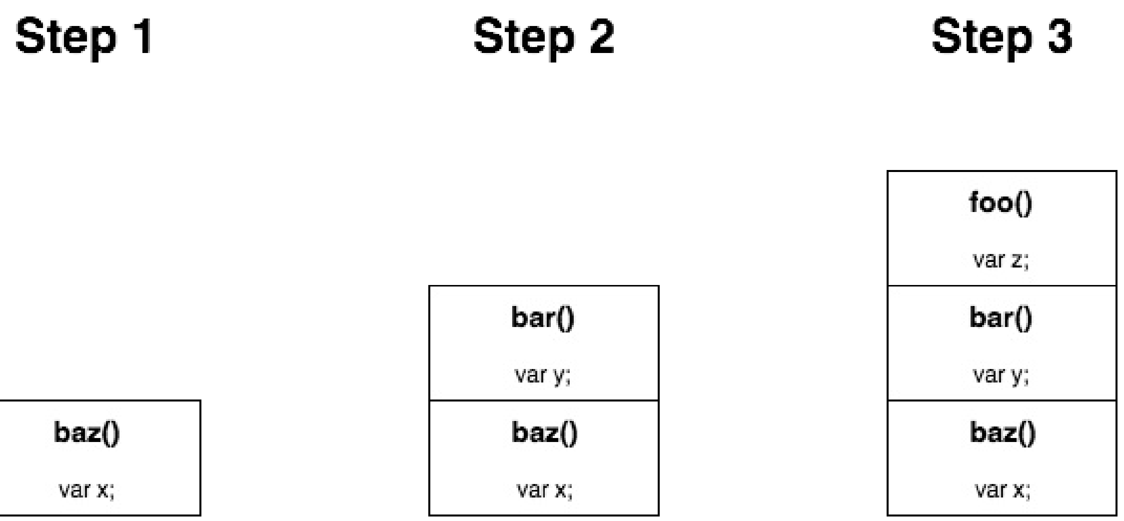
```
function foo() {
    var z = "foo!";
}

function bar() {
    var y = "bar!";
    foo();
}

function baz() {
    var x = "baz!";
    bar();
}

baz();
```

来一步步想象下这个程序的堆栈帧：



注意：如果这些函数间没有相互调用，而只是依次执行 -- 比如前一个函数运行结束后才开始调用下一个函数 `baz(); bar(); foo();` -- 则堆栈帧并没有产生；因为在下一个函数开始之前，上一个函数运行结束并把它的帧从堆栈里面移除了。

所以，每一个函数运行时候，都会占用一些内存。对多数程序来说，这没什么大不了的，不是吗？但是，一旦你引用了递归，问题就不一样了。 虽然你几乎肯定不会在一个调用栈中手动调用成千（或数百）次不同的函数，但你很容易看到产生数万个或更多递归调用的堆栈。

当引擎认为调用栈增加的太多并且应该停止增加时候，它会以主观的限制来阻止当前步骤，所以 `isOdd(..)` 或 `isEven(..)` 函数抛出了 `RangeError` 未知错误。这不太可能是内存接近零时候产生的限制，而是引擎的预测，因为如果这种程序持续运行下去，内存会爆掉的。由于引擎无法判断一个程序最终是否会停止，所以它必须做出确定的猜测。

引擎的限制因情况而定。规范里面并没有任何说明，因此，它也不是 必需的。但如果没有限制的话，设备很容易遭到破坏或恶意代码攻击，故而几乎所有的JS引擎都有一个限制。不同的设备环境、不同的引擎，会有不同的限制，也就无法预测或保证函数调用栈能调用多少次。

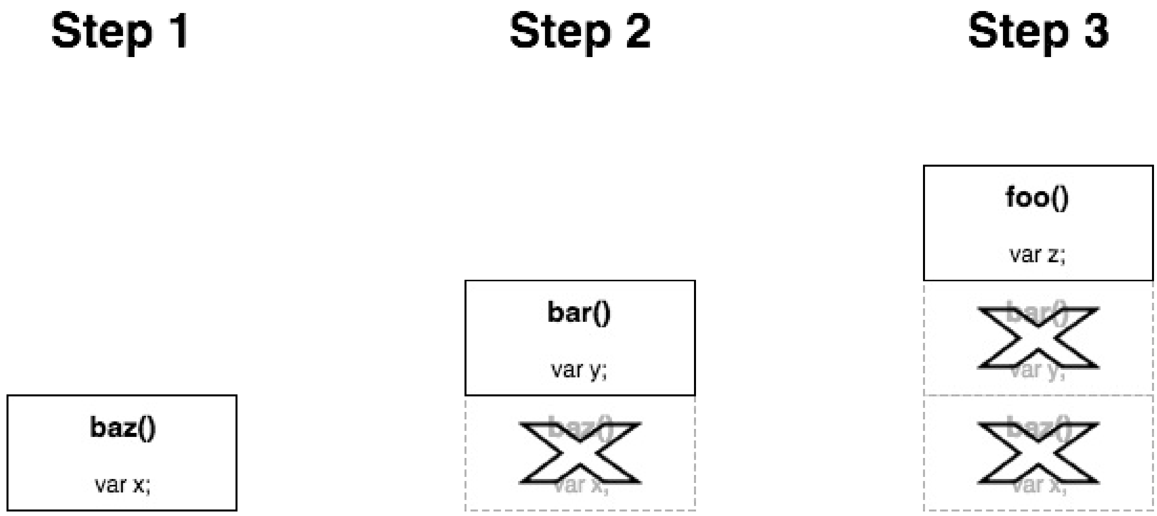
在处理大数据量时候，这个限制对于开发人员来说，会对递归的性能有一定的要求。我认为，这种限制也可能是造成开发人员不喜欢使用递归编程的最大原因。 遗憾的是，递归编程是一种编程思想而不是主流的编程技术。

尾调用

递归编程和内存限制都要比 JS 技术出现的早。追溯到上世纪 60 年代，当时开发人员想使用递归编程并希望运行在他们强大的计算机的设备，而所谓强大计算机的内存，尚远不如我们今天在手表上的内存。

幸运的是，在那个希望的原野上，进行了一个有力的观测。该技术称为 尾调用。

它的思路是如果一个回调从函数 `baz()` 转到函数 `bar()` 时候，而回调是在函数 `baz()` 的最底部执行 -- 也就是尾调用 -- 那么 `baz()` 的堆栈帧就不再需要了。也就意谓着，内存可以被回收，或只需简单的执行 `bar()` 函数。 如图所示：



尾调用并不是递归特有的；它适用于任何函数调用。但是，在大多数情况下，你的手动非递归调用栈不太可能超过 10 级，因此尾调用对你程序内存的影响可能相当低。

在递归的情况下，尾调用作用很明显，因为这意味着递归堆栈可以"永远"运行下去，唯一的性能问题就是计算，而不再是固定的内存限制。在固定的内存中尾递归可以运行 `O(1)` （常数阶时间复杂度计算）。

这些技术通常被称为尾调用优化（TCO），但重点在于从优化技术中，区分出在固定内存空间中检测尾调用运行的能力。从技术上讲，尾调用并不像大多数人所想的那样，它们的运行速度可能比普通回调还慢。TCO 是关于把尾调用更加高效运行的一些优化技术。

正确的尾调用 (PTC)

在 ES6 出来之前，JavaScript 对尾调用一直没明确规定（也没有禁用）。ES6 明确规定了 PTC 的特定形式，在 ES6 中，只要使用尾调用，就不会发生栈溢出。实际上这也就意味着，只要正确的使用 PTC，就不会抛出 `RangeError` 这样的异常错误。

首先，在 JavaScript 中应用 PTC，必须以严格模式书写代码。如果你以前没有用过严格模式，你得试着用用了。那么，您，应该已经在使用严格模式了吧！？

其次，正确 的尾调用就像这个样子：

```
return foo( .. );
```

换句话说，函数调用应该放在最后一步去执行，并且不管返回什么东东，都得有返回（ `return` ）。这样的话，JS 就不再需要当前的堆栈帧了。

下面这些 不能 称之为 PTC：

```
foo();
return;

// 或

var x = foo( .. );
return x;

// 或

return 1 + foo( .. );
```

注意：一些 JS 引擎 能够 把 `var x = foo(); return x;` 自动识别为 `return foo();`，这样也可以达到 PTC 的效果。但这毕竟不符合规范。

`foo(..)` 运行结束之后 `1+` 这部分才开始执行，所以此时的堆栈帧依然存在。

不过，下面这个 是 PTC：

```
return x ? foo( .. ) : bar( .. );
```

`x` 进行条件判断之后，或执行 `foo(..)`，或执行 `bar(..)`，不论执行哪个，返回结果都会被 `return` 返回掉。这个例子符合 PTC 规范。

为了避免堆栈增加，PTC 要求所有的递归必须是在尾部调用，因此，二分法递归 —— 两次（或以上）递归调用 —— 是不能实现 PTC 的。我们曾在文章的前面部分展示过把二分法递归转变为相互递归的例子。也许我们可以试着化整为零，把多重递归拆分成符合 PTC 规范的单个函数回调。

重构递归

如果你想用递归来处理问题，却又超出了 JS 引擎的内存堆栈，这时候就需要重构下你的递归调用，使它能够符合 PTC 规范（或着避免嵌套调用）。这里有一些重构方法也许可以用到，但需要根据实际情况权衡。

可读性强的代码，是我们的终级目标 —— 谨记，谨记。如果使用递归后会造成代码难以阅读/理解，那就 不要使用递归；换个容易理解的方法吧。

更换堆栈

对递归来说，最主要的问题是它的内存使用情况。保持堆栈帧跟踪函数调用的状态，并将其分派给下一个递归调用迭。如果我们弄清楚了如何重新排列我们的递归，就可以用 PTC 实现递归，并利用 JS 引擎对尾调用的优化处理，那么我们就不用在内存中保留当前的堆栈帧了。

来回顾下之前用到的一个求和的例子：

```
function sum(num1,...nums) {
    if (nums.length == 0) return num1;
    return num1 + sum( ...nums );
}
```

这个例子并不符合 PTC 规范。`sum(...nums)` 运行结束之后，`num1` 与 `sum(...nums)` 的运行结果进行了累加。这样的话，当其余参数 `...nums` 再次进行递归调用时候，为了得到其与 `num1` 累加的结果，必须要保留上一次递归调用的堆栈帧。

重构策略的关键点在于，我们可以通过把 置后 处理累加改为 提前 处理，来消除对堆栈的依赖，然后将该部分结果作为参数传递到递归调用。换句话说，我们不用在当前运用函数的堆栈帧中保留 `num1 + sum(...num1)` 的总和，而是把它传递到下一个递归的堆栈帧中，这样就能释放当前递归的堆栈帧。

开始之前，我们做些改动：把部分结果作为一个新的第一个参数传入到函数 `sum(..)`：

```
function sum(result,num1,...nums) {
    // ..
}
```

这次我们先把 `result` 和 `num1` 提前计算，然后把 `result` 作为参数一起传入：

```
"use strict";

function sum(result,num1,...nums) {
    result = result + num1;
    if (nums.length == 0) return result;
    return sum( result, ...nums );
}
```

现在 `sum(..)` 已经符合 PTC 优化规范了！耶！

但是还有一个缺点，我们修改了函数的参数传递形式后，用法就跟以前不一样了。调用者不得不在需要求和的那些参数的前面，再传递一个 `0` 作为第一个参数。

```
sum( /*initialResult=*/0, 3, 1, 17, 94, 8 );        // 123
```

这就尴尬了。

通常，大家的处理方式是，把这个尴尬的递归函数重新命名，然后定义一个接口函数把问题隐藏起来：

```
"use strict";

function sumRec(result,num1,...nums) {
    result = result + num1;
    if (nums.length == 0) return result;
    return sumRec( result, ...nums );
}

function sum(...nums) {
    return sumRec( /*initialResult=*/0, ...nums );
}

sum( 3, 1, 17, 94, 8 );                              // 123
```

情况好了些。但依然有问题：之前只需要一个函数就能解决的事，现在我们用了两个。有时候你会发现，在处理这类问题上，有些开发者用内部函数把递归“藏了起来”：

```
"use strict";

function sum(...nums) {
    return sumRec( /*initialResult=*/0, ...nums );

    function sumRec(result,num1,...nums) {
        result = result + num1;
        if (nums.length == 0) return result;
        return sumRec( result, ...nums );
    }
}

sum( 3, 1, 17, 94, 8 );                              // 123
```

这个方法的缺点是，每次调用外部函数 `sum(..)` ，我们都得重新创建内部函数 `sumRec(..)` 。我们可以把他们平级放置在立即执行的函数中，只暴露出我们想要的那个的函数：

```
"use strict";

var sum = (function IIFE(){

    return function sum(...nums) {
        return sumRec( /*initialResult=*/0, ...nums );
    }

    function sumRec(result,num1,...nums) {
        result = result + num1;
        if (nums.length == 0) return result;
        return sumRec( result, ...nums );
    }

})();

sum( 3, 1, 17, 94, 8 );                             // 123
```

好啦，现在即符合了 PTC 规范，又保证了 `sum(..)` 参数的整洁性，调用者不需要了解函数的内部实现细节。完美！

可是...天呐，本来是简单的递归函数，现在却出现了很多噪点。可读性已经明显降低。至少说，这是不成功的。有些时候，这只是我们能做的最好的。

幸运的事，在一些其它的例子中，比如上一个例子，有一个比较好的方式。一起重新看下：

```
"use strict";

function sum(result,num1,...nums) {
    result = result + num1;
    if (nums.length == 0) return result;
    return sum( result, ...nums );
}

sum( /*initialResult=*/0, 3, 1, 17, 94, 8 );        // 123
```

也许你会注意到，`result` 就像 `num1` 一样，也就是说，我们可以把列表中的第一个数字作为我们的运行总和；这甚至包括了第一次调用的情况。我们需要的是重新命名这些参数，使函数清晰可读：

```
"use strict";

function sum(num1,num2,...nums) {
    num1 = num1 + num2;
    if (nums.length == 0) return num1;
    return sum( num1, ...nums );
}

sum( 3, 1, 17, 94, 8 );                             // 123
```

帅呆了。比之前好了很多，嗯？！我认为这种模式在声明/合理和执行之间达到了很好的平衡。

让我们试着重构下前面的 `maxEven(..)`（目前还不符合 PTC 规范）。就像之前我们把参数的和作为第一个参数一样，我们可以依次减少列表中的数字，同时一直把遇到的最大偶数作为第一个参数。

为了清楚起见，我们可能使用算法策略（类似于我们之前讨论过的）：

1. 首先对前两个参数 `num1` 和 `num2` 进行对比。
2. 如果 `num1` 是偶数，并且 `num1` 大于 `num2`，`num1` 保持不变。
3. 如果 `num2` 是偶数，把 `num2` 赋值给 `num1`。
4. 否则的话，`num1` 等于 `undefined`。
5. 如果除了这两个参数之外，还存在其它参数 `nums`，把它们与 `num1` 进行递归对比。
6. 最后，不管是什么值，只需返回 `num1`。

依照上面的步骤，代码如下：

```
"use strict";

function maxEven(num1,num2,...nums) {
    num1 =
        (num1 % 2 == 0 && !(maxEven( num2 ) > num1)) ?
            num1 :
            (num2 % 2 == 0 ? num2 : undefined);

    return nums.length == 0 ?
        num1 :
        maxEven( num1, ...nums )
}
```

注意： 函数第一次调用 `maxEven(..)` 并不是为了 PTC 优化，当它只传递 `num2` 时，只递归一级就返回了；它只是一个避免重复 `%` 逻辑的技巧。因此，只要该调用是完全不同的函数，就不会增加递归堆栈。第二次调用 `maxEven(..)` 是基于 PTC 优化角度的真正递归调用，因此不会随着递归的进行而造成堆栈的增加。

重申下，此示例仅用于说明将递归转化为符合 PTC 规范以优化堆栈（内存）使用的方法。求最大偶数值的更直接方法可能是，先对参数列表中的 `nums` 过滤，然后冒泡或排序处理。

基于 PTC 重构递归，固然对简单的声明形式有一些影响，但依然有理由去做这样的事。不幸的是，存在一些递归，即使我们使用了接口函数来扩展，也不会很好，因此，我们需要有不同的思路。

后继传递格式（CPS）

在 JavaScript 中， *continuation* 一词通常用于表示在某个函数完成后指定需要执行的下一个步骤的回调函数。组织代码，使得每个函数在其结束时接收另一个执行函数，被称为后继传递格式（CPS）。

有些形式的递归，实际上是无法按照纯粹的 PTC 规范重构的，特别是相互递归。我们之前提到过的 `fib(..)` 函数，以及我们派生出来的相互递归形式。这两个情况，皆是存在多个递归调用，这些递归调用阻碍了 PTC 内存优化。

但是，你可以执行第一个递归调用，并将后续递归调用包含在后续函数中并传递到第一个调用。尽管这意味着最终需要在堆栈中执行更多的函数，但由于后继函数所包含的都是 PTC 形式的，所以堆栈内存的使用情况不会无限增长。

把 `fib(..)` 做如下修改：

```
"use strict";

function fib(n,cont = identity) {
    if (n <= 1) return cont( n );
    return fib(
        n - 2,
        n2 => fib(
            n - 1,
            n1 => cont( n2 + n1 )
        )
    );
}
```

仔细看下都做了哪些事情。首先，我们默认用了第三章中的 `cont(..)` 后继函数表示 `identity(..)`；记住，它只简单的返回传递给它的任何东西。

更重要的是，这里面增加了不仅仅是一个而是两个后续函数。第一个后续函数接收 `fib(n-2)` 的运行结果作为参数 `n2`。第二个内部后续函数接收 `fib(n-1)` 的运行结果作为参数 `n1`。当得到 `n1` 和 `n2` 的值后，两者再相加（`n2 + n1`），相加的运行结果会传入到下一个后续函数 `cont(..)`。

也许这将有助于我们梳理下流程：就像我们之前讨论的，在递归堆栈之后，当我们传递部分结果而不是返回它们时，每一步都被包含在一个后续函数中，这拖慢了计算速度。这个技巧允许我们执行多个符合 PTC 规范的步骤。

在静态语言中，CPS通常为尾调用提供了编译器可以自动识别并重新排列递归代码以利用的机会。很可惜，不能用在原生 JS 上。

在 JavaScript 中，你得自己书写出符合 CPS 格式的代码。这并不是明智的做法；以命令符号声明的形式肯定会让内容有些不清楚。 但总的来说，这种形式仍然要比 `for` 循环更具有声明性。

警告： 我们需要注意的一个比较重要的事项是，在 CPS 中，创建额外的内部后续函数仍然消耗内存，但有些不同。并不是之前的堆栈帧累积，闭包只是消耗多余的内存空间（一般情况下，是堆栈里面的多余内存空间）。在这些情况下，引擎似乎没有启动 `RangeError` 限制，但这并不意味着你的内存使用量是按比例固定好的。

弹簧床

除了 CPS 后续传递格式之外，另外一种内存优化的技术称为弹簧床。在弹簧床格式的代码中，同样的创建了类似 CPS 的后续函数，不同的是，它们没有被传递，而是被简单的返回了。

不再是函数调用另外的函数，堆栈的深度也不会大于一层，因为每个函数只会返回下一个将调用的函数。循环只是继续运行每个返回的函数，直到再也没有函数可运行。

弹簧床的优点之一是在非 PTC 环境下你一样可以应用此技术。另一个优点是每个函数都是正常调用，而不是 PTC 优化，所以它可以运行得更快。

一起来试下 `trampoline(..)` ：

```
function trampoline(fn) {
    return function trampolined(...args) {
        var result = fn( ...args );

        while (typeof result == "function") {
            result = result();
        }

        return result;
    };
}
```

当返回一个函数时，循环继续，执行该函数并返回其运行结果，然后检查返回结果的类型。一旦返回的结果类型不是函数，弹簧床就认为函数调用完成了并返回结果值。

所以我们可能需要使用前面讲到的，将部分结果作为参数传递的技巧。以下是我们在之前的数组求和中使用此技巧的示例：

```
var sum = trampoline(
    function sum(num1,num2,...nums) {
        num1 = num1 + num2;
        if (nums.length == 0) return num1;
        return () => sum( num1, ...nums );
    }
);

var xs = [];
for (let i=0; i<20000; i++) {
    xs.push( i );
}

sum( ...xs );                    // 199990000
```

缺点是你需要将递归函数包裹在执行弹簧床功能的函数中; 此外，就像 CPS 一样，需要为每个后续函数创建闭包。然而，与 CPS 不一样的地方是，每个返回的后续数数，运行并立即完成，所以，当调用堆栈的深度用尽时，引擎中不会累积越来越多的闭包。

除了执行和记忆性能之外，弹簧床技术优于CPS的优点是它们在声明递归形式上的侵入性更小，由于你不必为了接收后续函数的参数而更改函数参数，所以除了执行和内存性能之外，弹簧床技术优于 CPS 的地方还有，它们在声明递归形式上侵入性更小。虽然弹簧床技术并不是理想的，但它们可以有效地在命令循环代码和声明性递归之间达到平衡。

总结

递归，是指函数递归调用自身。呃，这就是递归的定义。明白了吧！？

直递归是指对自身至少调用一次，直到满足基本条件才能停止调用。多重递归（像二分递归）是指对自身进行多次调用。相互递归是当两个或以上函数循环递归 相互 调用。而递归的优点是它更具声明性，因此通常更易于阅读。

递归的优点是它更具声明性，因此通常更易于阅读。缺点通常是性能方面，但是相比执行速度，更多的限制在于内存方面。

尾调用是通过减少或释放堆栈帧来节约内存空间。要在 JavaScript 中实现尾调用 “优化”，需要基于严格模式和适当的尾调用（ PTC ）。我们也可以混合几种技术来将非 PTC 递归函数重构为 PTC 格式，或者至少能通过平铺堆栈来节约内存空间。

谨记：递归应该使代码更容易读懂。如果你误用或滥用递归，代码的可读性将会比命令形式更糟。千万不要这样做。

JavaScript 轻量级函数式编程

第 10 章：异步的函数式

阅读到这里，你已经学习了我所说的所有轻量级函数式编程的基础概念，在本章节中，我们将把这些概念应有到不同的情景当中，但绝对不会有新的知识点。

到目前为止，我们所说的一切都是同步的，意味着我们调用函数，传入参数后马上就会得到返回值。大部分的情况下是没问题的，但这几乎满足不了现有的 JS 应用。为了能在当前的 JS 环境里使用上函数式编程，我们需要去了解异步的函数式编程。

本章的目的是拓展我们对用函数式编程管理数据的思维，以便之后我们在更多的业务上应用。

时间状态

在你所有的应用里，最复杂的状态就是时间。当你操作的数据状态改变过程比较直观的时候，是很容易管理的。但是，如果状态随着时间因为响应事件而隐晦的变化，管理这些状态的难度将会成几何级增长。

我们在本文中介绍的函数式编程可以让代码变得更可读，从而增强了可靠性和可预见性。但是当你添加异步操作到你的项目里的时候，这些优势将会大打折扣。

必须明确的一点是：并不是说一些操作不能用同步来完成，或者触发异步行为很容易。协调那些可能会改变应用程序的状态的响应，这需要大量额外的工作。

所以，作为作者的你最好付出一些努力，或者只是留给阅读你代码的人一个难题，去弄清楚如果 A 在 B 之前完成，项目中状态是什么，还有相反的情况是什么？这是一个浮夸的问题，但以我的观点来看，这有一个确切的答案：如果可以把复杂的代码变得更容易理解，作者就必须花费更多心思。

减少时间状态

异步编程最为重要的一点是通过抽象时间来简化状态变化的管理。

为说明这一点，让我们先来看下一种有竞争状态（又称，时间复杂度）的糟糕情况，且必须手动去管理里面的状态：

```
var customerId = 42;
var customer;

lookupCustomer( customerId, function onCustomer(customerRecord){
    var orders = customer ? customer.orders : null;
    customer = customerRecord;
    if (orders) {
        customer.orders = orders;
    }
} );

lookupOrders( customerId, function onOrders(customerOrders){
    if (!customer) {
        customer = {};
    }
    customer.orders = customerOrders;
} );
```

回调函数 `onCustomer(..)` 和 `onOrders(..)` 之间是互为竞争关系。假设他们都在运行，两者都有可能先运行，那将无法预测到会发生什么。

如果我们可以把 `lookupOrders(..)` 写到 `onCustomer(..)` 里面，那我们就可以确认 `onOrders(..)` 会在 `onCustomer(..)` 之后运行，但我们不能这么做，因为我们需要让 2 个查询同时执行。

所以，为了让这个基于时间的复杂状态正常化，我们用相应的 `if`-声明在各自的回调函数里来检查外部作用域的变量 `customer`。当各自的回调函数被执行，将会去检测 `customer` 的状态，从而确定各自的执行顺序，如果 `customer` 在回调函数里还没被定义，那他就是先运行的，否则则是第二个运行的。

这些代码可以运行，但是他违背了可读性的原则。时间复杂度让这个代码变得难以阅读。

让我们改用 JS promise 来把时间因素抽离出来：

```
var customerId = 42;

var customerPromise = lookupCustomer( customerId );
var ordersPromise = lookupOrders( customerId );

customerPromise.then( function onCustomer(customer){
    ordersPromise.then( function onOrders(orders){
        customer.orders = orders;
    } );
} );
```

现在 `onOrders(..)` 回调函数存在 `onCustomer(..)` 回调函数里，所以他们各自的执行顺序是可以保证的。在各自的 `then(..)` 运行之前 `lookupCustomer(..)` 和 `lookupOrders(..)` 被分别的调用，两个查询就已经并行的执行完了。

这可能不太明显，但是这个代码里还有其他内在的竞争状态，那就是 promise 的定义没有被体现出来。如果 `orders` 的查询在把 `onOrders(..)` 回调函数被 `ordersPromise.then(..)` 调用前完成，那么就需要一些比较智能的 东西 来保存 `orders` 直到 `onOrders(..)` 能被调用。同理，`record` （或者说`customer` ）对象是否能在 `onCustomer(..)` 执行时被接收到。

这里的 东西 和我们之前讨论过的时间复杂度类似。但我们不必去担心这些复杂性，无论是编码或者是读（更为重要）这些代码的时候，因为对我们来说，promise 所处理的就是时间复杂度上的问题。

promise 以时间无关的方式来作为一个单一的值。此外，获取 promise 的返回值是异步的，但却是通过同步的方法来赋值。或者说， promise 给 `=` 操作符扩展随时间动态赋值的功能，通过可靠的（时间无关）方式。

接下来我们将探索如何以相同的方式，在时间上异步地拓展本书之前同步的函数式编程操作。

积极的 vs 惰性的

积极的和惰性的在计算机科学的领域并不是表扬或者批评的意思，而是描述一个操作是否立即执行或者是延时执行。

我们在本例子中看到的函数式编程操作可以被称为积极的，因为它们同步（即时）地操作着离散的即时值或值的列表/结构上的值。

回忆下：

```
var a = [1,2,3]

var b = a.map( v => v * 2 );

b;            // [2,4,6]
```

这里 `a` 到 `b` 的映射就是积极的，因为它在执行的那一刻映射了数组 `a` 里的所有的值，然后生成了一个新的数组 `b` 。即使之后你去修改 `a` ，比如说添加一个新的值到数组的最后一位，也不会影响到 `b` 的内容。这就是积极的函数式编程。

但是如果是一个惰性的函数式编程操作呢？思考如下情况：

```
var a = [];

var b = mapLazy( a, v => v * 2 );

a.push( 1 );

a[0];         // 1
b[0];         // 2

a.push( 2 );

a[1];         // 2
b[1];         // 4
```

我们可以想象下 `mapLazy(..)` 本质上“监听”了数组 `a`，只要一个新的值添加到数组的末端（使用 `push(..)`），它都会运行映射函数 `v => v * 2` 并把改变后的值添加到数组 `b` 里。

注意： `mapLazy(..)` 的实现没有被写出来，是因为它是虚构的方法，是不存在的。如果要实现 `a` 和 `b` 之间的惰性的操作，那么简单的数组就需要变得更加聪明。

考虑下把 `a` 和 `b` 关联到一起的好处，无论何时何地，你添加一个值进 `a` 里，它都将改变且映射到 `b` 里。它比同为声明式函数式编程的 `map(..)` 更强大，但现在它可以随时地变化，进行映射时你不用知道 `a` 里面所有的值。

响应式函数式编程

为了理解如何在2个值之间创建和使用惰性的映射，我们需要去抽象我们对列表（数组）的想法。

让我们来想象一个智能的数组，不只是简单地获得值，还是一个懒惰地接受和响应（也就是“反应”）值的数组。考虑下：

```
var a = new LazyArray();

var b = a.map( function double(v){
    return v * 2;
} );

setInterval( function everySecond(){
    a.push( Math.random() );
}, 1000 );
```

至此，这段代码的数组和普通的没有什么区别。唯一不同的是在我们执行 `map(..)` 来映射数组 `a` 生成数组 `b` 之后，定时器在 `a` 里面添加随机的值。

但是这个虚构的 `LazyArray` 有点不同，它假设了值可以随时的一个一个添加进去。就像随时可以 `push(..)` 你想要的值一样。可以说 `b` 就是一个惰性映射 `a` 最终值的数组。

此外，当 `a` 或者 `b` 改变时，我们不需要确切地保存里面的值，这个特殊的数组将会保存它所需的值。所以这些数组不会随着时间而占用更多的内存，这是 惰性数据结构和懒操作的重要特点。事实上，它看起来不像数组，更像是buffer（缓冲区）。

普通的数组是积极的，所以它会立马保存所有它的值。"惰性数组" 的值则会延迟保存。

由于我们不一定要知道 `a` 什么时候添加了新的值，所以另一个关键就是我们需要有去监听 `b` 并在有新值的时候通知它的能力。我们可以想象下监听器是这样的：

```
b.listen( function onValue(v){
    console.log( v );
} );
```

`b` 是反应性的，因为它被设置为当 `a` 有值添加时进行反应。函数式编程操作当中的 `map(..)` 是把数据源 `a` 里面的所有值转移到目标 `b` 里。每次映射操作都是我们使用同步函数式编程进行单值建模的过程，但是接下来我们将让这种操作变得可以响应式执行。

注意： 最常用到这些函数式编程的是响应式函数式编程(FRP)。我故意避开这个术语是因为一个有关于 FP + Reactive 是否真的构成 FRP 的辩论。我们不会全面深入了解 FRP 的所有含义，所以我会继续称之为响应式函数式编程。或者，如果你不会感觉那么困惑，也可以称之为事件机制函数式编程。

我们可以认为 `a` 是生成值的而 `b` 则是去消费这些值的。所以为了可读性，我们得重新整理下这段代码，让问题归结于 生产者 和 消费者。

```
// 生产者:

var a = new LazyArray();

setInterval( function everySecond(){
    a.push( Math.random() );
}, 1000 );


// **************************
// 消费者:

var b = a.map( function double(v){
    return v * 2;
} );

b.listen( function onValue(v){
    console.log( v );
} );
```

`a` 是一个行为本质上很像数据流的生产者。我们可以把每个值赋给 `a` 当作一个事件。`map(..)` 操作会触发 `b` 上面的 `listen(..)` 事件来消费新的值。

我们分离 生产者 和 消费者 的相关代码，是因为我们的代码应该各司其职。这样的代码组织可以很大程度上提高代码的可读性和维护性。

声明式的时间

我们应该非常谨慎地讨论如何介绍时间状态。具体来说，正如 promise 从单个异步操作中抽离出我们所担心的时间状态，响应式函数式编程从一系列的值/操作中抽离（分割）了时间状态。

从 `a` （生产者）的角度来说，唯一与时间相关的就是我们手动调用的 `setInterval(..)` 循环。但它只是为了示范。

想象下 `a` 可以被绑定上一些其他的事件源，比如说用户的鼠标点击事件和键盘按键事件，服务端来的 websocket 消息等。在这些情况下，`a` 没必要关注自己的时间状态。每当值准备好，它就只是一个与值连接的无时态管道。

从 `b` （消费者）的角度来说，我们不用知道或者关注 `a` 里面的值在何时何地来的。事实上，所有的值都已经存在。我们只关注是否无论何时都能取到那些值。或者说，`map(..)` 的转换操作是一个无时态（惰性）的建模过程。

时间 与 `a` 和 `b` 之间的关系是声明式的，不是命令式的。

以 operations-over-time 这种方式来组织值可能不是很有效。让我们来对比下相同的功能如何用命令式来表示：

```
// 生产者:

var a = {
    onValue(v){
        b.onValue( v );
    }
};

setInterval( function everySecond(){
    a.onValue( Math.random() );
}, 1000 );


// **************************
// 消费者:

var b = {
    map(v){
        return v * 2;
    },
    onValue(v){
        v = this.map( v );
        console.log( v );
    }
};
```

这似乎很微妙，但这就是存在于命令式版本的代码和之前声明式的版本之间一个很重要的不同点，除了 `b.onValue(..)` 需要自己去调用 `this.map(..)` 之外。在之前的代码中， `b` 从 `a` 当中去拉取，但是在这个代码中，`a` 推送给 `b` 。换句话说，把 `b = a.map(..)` 替换成 `b.onValue(v)` 。

在上面的命令式代码中，以消费者的角度来说它并不清楚 `v` 从哪里来。此外命令式强硬的把代码 `b.onValue(..)` 夹杂在生产者 `a` 的逻辑里，这有点违反了关注点分离原则。这将会让分离生产者和消费者变得困难。

相比之下，在之前的代码中，`b = a.map(..)` 表示了 `b` 的值来源于 `a` ，对于如同抽象事件流的数据源 `a` ，我们不需要关心。我们可以 确信 任何来自于 `a` 到 `b` 里的值都会通过 `map(..)` 操作。

映射之外的东西

为了方便，我们已经说明了通过随着时间一次一次的用 `map(..)` 来绑定 `a` 和 `b` 的概念。其实我们许多其他的函数式编程操作也可以做到这种效果。

思考下：

```
var b = a.filter( function isOdd(v) {
    return v % 2 == 1;
} );

b.listen( function onlyOdds(v){
    console.log( "Odd:", v );
} );
```

这里可以看到 `a` 的值肯定会通过 `isOdd(..)` 赋值给 `b`。

即使是 `reduce(..)` 也可以持续的运行：

```
var b = a.reduce( function sum(total,v){
    return total + v;
} );

b.listen( function runningTotal(v){
    console.log( "New current total:", v );
} );
```

因为我们调用 `reduce(..)` 是没有给具体 `initialValue` 的值，无论是 `sum(..)` 或者 `runningTotal(..)` 都会等到有 2 个来自 `a` 的参数时才会被调用。

这段代码暗示了在 reduction 里面有一个 内存空间， 每当有新的值进来的时候，`sum(..)` 才会带上第一个参数 `total` 和第二个参数 `v` 被调用。

其他的函数式编程操作会在内部作用域请求一个缓存区，比如说 `unique(..)` 可以追踪每一个它访问过的值。

Observables

希望现在你可以察觉到响应式，事件式，类数组结构的数据的重要性，就像我们虚构出来的 `LazyArray` 一样。值得高兴的是，这类的数据结构已经存在的了，它就叫 observable。

注意： 只是做些假设（希望）：接下来的讨论只是简要的介绍 observables。这是一个需要我们花时间去探究的深层次话题。但是如果你理解本文中的轻量级函数式编程，并且知道如何通过函数式编程的原理来构建异步的话，那么接着学习 observables 将会变得得心应手。

现在已经有各种各样的 Observables 的库类， 最出名的是 RxJS 和 Most。在写这篇文章的时候，正好有一个直接向 JS 里添加 observables 的建议，就像 promise。为了演示，我们将用 RxJS 风格的 Observables 来完成下面的例子。

这是我们一个较早的响应式的例子，但是用 Observables 来代替 `LazyArray`：

```
// 生产者:

var a = new Rx.Subject();

setInterval( function everySecond(){
    a.next( Math.random() );
}, 1000 );


// **************************
// 消费者:

var b = a.map( function double(v){
    return v * 2;
} );

b.subscribe( function onValue(v){
    console.log( v );
} );
```

在 RxJS 中，一个 Observer 订阅一个 Observable。如果你把 Observer 和 Observable 的功能结合到一起，那就会得到一个 Subject。因此，为了保持代码的简洁，我们把 `a` 构建成一个 Subject，所以我们可以调用它的 `next(..)` 方法来添加值（事件）到他的数据流里。

如果我们要让 Observer 和 Observable 保持分离：

```
// 生产者:

var a = Rx.Observable.create( function onObserve(observer){
    setInterval( function everySecond(){
        observer.next( Math.random() );
    }, 1000 );
} );
```

在这个代码里，`a` 是 Observable，毫无疑问，`observer` 就是独立的 observer，它可以去“观察”一些事件（比如我们的 `setInterval(..)` 循环），然后我们使用它的 `next(..)` 方法来发送一些事件到 observable `a` 的流里。

除了 `map(..)`，RxJS 还定义了超过 100 个可以在有新值添加时才触发的方法。就像数组一样。每个 Observable 的方法都会返回一个新的 Observable，意味着他们是链式的。如果一个方法被调用，则它的返回值应该由输入的 Observable 去返回，然后触发到输出的 Observable里，否则抛弃。

一个链式的声明式 observable 的例子：

```
var b =
    a
    .filter( v => v % 2 == 1 )        // 过滤掉偶数
    .distinctUntilChanged()            // 过滤连续相同的流
    .throttle( 100 )                  // 函数节流（合并100毫秒内的流）
    .map( v = v * 2 );                // 变2倍

b.subscribe( function onValue(v){
    console.log( "Next:", v );
} );
```

注意： 这里的链式写法不是一定要把 observable 赋值给 `b` 和调用 `b.subscribe(..)` 分开写，这样做只是为了让每个方法都会得到一个新的返回值。通常，`subscribe(..)` 方法都会在链式写法的最后被调用。

总结

这本书详细的介绍了各种各样的函数式编程操作，例如：把单个值（或者说是一个即时列表的值）转换到另一个值里。

对于那些有时态的操作，所有基础的函数式编程原理都可以无时态的应用。就像 promise 创建了一个单一的未来值，我们可以创建一个积极的列表的值来代替像惰性的observable（事件）流的值。

数组的 `map(..)` 方法会用当前数组中的每一个值运行一次映射函数，然后放到返回的数组里。而 observable 数组里则是为每一个值运行一次映射函数，无论这个值何时加入，然后把它返回到 observable 里。

或者说，如果数组对函数式编程操作是一个积极的数据结构，那么 observable 相当于持续惰性的。

JavaScript 轻量级函数式编程

第 11 章：融会贯通

现在你已经掌握了所有需要掌握的关于 JavaScript 轻量级函数式编程的内容。下面不会再引入新的概念。

本章主要目标是概念的融会贯通。通过研究代码片段，我们将本书中大部分主要概念联系起来并学以致用。

建议进行大量深入的练习来熟悉这些技巧，因为理解本章内容对于将来你在实际编程场景中应用函数式编程原理至关重要。

准备

我们来写一个简单的股票行情工具吧。

注意：可以在本书的 GitHub 仓库（https://github.com/getify/Functional-Light-JS）下的 `ch11-code/` 目录里找到参考代码。同时，在书中讨论到的函数式编程辅助函数的基础上，我们筛选了所需的一部分放到了 `ch11-code/fp-helpers.js` 文件中。本章中，我们只会讨论到其中相关的部分。

首先来编写 HTML 部分，这样便可以对信息进行展示了。我们在 `ch11-code/index.html` 文件中先写一个空的 `<ul ..>` 元素，在运行时，DOM 会被填充成：

```html
<ul id="stock-ticker">
    <li class="stock" data-stock-id="AAPL">
        <span class="stock-name">AAPL</span>
        <span class="stock-price">$121.95</span>
        <span class="stock-change">+0.01</span>
    </li>
    <li class="stock" data-stock-id="MSFT">
        <span class="stock-name">MSFT</span>
        <span class="stock-price">$65.78</span>
        <span class="stock-change">+1.51</span>
    </li>
    <li class="stock" data-stock-id="GOOG">
        <span class="stock-name">GOOG</span>
        <span class="stock-price">$821.31</span>
        <span class="stock-change">-8.84</span>
    </li>
</ul>
```

我必须要事先提醒你的一点是，和 DOM 进行交互属于输入／输出操作，这也意味着会产生一定的副作用。我们不能消除这些副作用，所以我们尽量减少和 DOM 相关的操作。这些技巧在第 5 章中已经提到了。

概括一下我们的小工具的功能：代码将在每次收到添加新股票事件时添加 `<li ..>` 元素，并在股票价格更新事件发生时更新价格。

在第 11 章的示例代码 `ch11-code/mock-server.js` 中，我们设置了一些定时器，把随机生成的假股票数据推送到一个简单的事件发送器中，来模拟从服务器收到的股票数据。我们暴露了一个 `connectToServer()` 接口来实现模拟，但是实际上，它只是返回了一个假的事件发送器。

注意： 这个文件是用来模拟数据的，所以我没有花费太多的精力让它完全符合函数式编程，不建议大家花太多时间研究这个文件中的代码。如果你写了一个真正的服务器 —— 对于那些雄心勃勃的读者来说，这是一个有趣的加分练习 —— 这时你才应该考虑采用函数式编程思想来实现这些代码。

我们在 `ch11-code/stock-ticker-events.js` 中，创建了一些 observable（通过 RxJS）连接到事件发送器对象上。通过调用 `connectToServer()` 来获取这个事件的发射器，然后监听名称为 `"stock"` 的事件，通过这个事件来添加一个新的股票代码，同时监听名称为 `"stock-update"` 的事件，通过这个事件来更新股票价格和涨跌幅。最后，我们定义一些转换函数，来对这些 observable 传入的数据进行格式化。

在 `ch11-code/stock-ticker.js` 中，我们将我们的界面操作（DOM 部分的副作用）定义在 `stockTickerUI` 对象的方法中。我们还定义了各种辅助函数，包括 `getElemAttr(..)`，`stripPrefix(..)` 等等。最后，我们通过 `subscribe(..)` 监听两个 observable，来获得格式化好的数据，渲染到 DOM 上。

股票信息

一起看看 `ch11-code/stock-ticker-events.js` 中的代码，我们先从一些基本的辅助函数开始：

```
function addStockName(stock) {
    return setProp( "name", stock, stock.id );
}
function formatSign(val) {
    if (Number(val) > 0) {
        return `+${val}`;
    }
    return val;
}
function formatCurrency(val) {
    return `$${val}`;
}
function transformObservable(mapperFn,obsv){
    return obsv.map( mapperFn );
}
```

这些纯函数应该很容易理解。参见第 4 章 `setProp(..)` 在设置新属性之前复制了对象。这实践到了我们在第 6 章中学习到的原则：通过把变量当作不可变的变量来避免副作用，即使其本身是可变的。

`addStockName(..)` 用来在股票信息对象中添加一个 `name` 属性，它的值和这个对象 `id` 一致。`name` 会作为股票的名称展示在工具中。

有一个关于 `transformObservable(..)` 的颇为微妙的注意事项：表面上看起来在 `map(..)` 函数中返回一个新的 observable 是纯函数操作，但是事实上，`obsv` 的内部状态被改变了，这样才能够和 `map(..)` 返回的新的 observable 连接起来。这个副作用并不是个大问题，而且不会影响我们的代码可读性，但是随时发现潜在的副作用是非常重要的，这样就不会在出错时倍感惊讶！

当从“服务器”获取股票信息时，数据是这样的：

```
{ id: "AAPL", price: 121.7, change: 0.01 }
```

在把 `price` 的值显示到 DOM 上之前，需要用 `formatCurrency(..)` 函数格式化一下（比如变成 `"$121.70"`），同时需要用 `formatChange(..)` 函数格式化 `change` 的值（比如变成 `"+0.01"`）。但是我们不希望修改消息对象中的 `price` 和 `change`，所以我们需要一个辅助函数来格式化这些数字，并且要求这个辅助函数返回一个新的消息对象，其中包含格式化好的 `price` 和 `change`：

```
function formatStockNumbers(stock) {
    var updateTuples = [
        [ "price", formatPrice( stock.price ) ],
        [ "change", formatChange( stock.change ) ]
    ];

    return reduce( function formatter(stock,[propName,val]){
        return setProp( propName, stock, val );
    } )
    ( stock )
    ( updateTuples );
}
```

我们创建了 `updateTuples` 元组来保存 `price` 和 `change` 的信息，包括属性名称和格式化好的值。把 `stock` 对象作为 `initialValue`，对元组进行 `reduce(..)`（参考第 8 章）。把元组中的信息解构成 `propName` 和 `val`，然后返回了 `setProp(..)` 调用的结果，这个结果是一个被复制了的新的对象，其中的属性被修改过了。

下面我们再定义几个辅助函数：

```
var formatDecimal = unboundMethod( "toFixed" )( 2 );
var formatPrice = pipe( formatDecimal, formatCurrency );
var formatChange = pipe( formatDecimal, formatSign );
var processNewStock = pipe( addStockName, formatStockNumbers );
```

`formatDecimal(..)` 函数接收一个数字作为参数（如 `2.1`）并且调用数字的 `toFixed( 2 )` 方法。我们使用了第 8 章介绍的 `unboundMethod(..)` 来创建一个独立的延迟绑定函数。

`formatPrice(..)`，`formatChange(..)` 和 `processNewStock(..)` 都用到了 `pipe(..)` 来从左到右地组合运算（见第 4 章）。

为了能在事件发送器的基础上创建 observable（见第 10 章），我们将封装一个独立的柯里化辅助函数（见第 3 章）来包装 RxJS 的 `Rx.Observable.fromEvent(..)`：

```
var makeObservableFromEvent = curry( Rx.Observable.fromEvent, 2 )( server );
```

这个函数特定地监听了 `server`（事件发送器），在接受了事件名称字符串参数后，就能生成 observable 了。我们准备好了创建 observer 的所有代码片段后，用映射函数转换 observer 来格式化获取到的数据：

```
var observableMapperFns = [ processNewStock, formatStockNumbers ];

var [ newStocks, stockUpdates ] = pipe(
    map( makeObservableFromEvent ),
    curry( zip )( observableMapperFns ),
    map( spreadArgs( transformObservable ) )
)
( [ "stock", "stock-update" ] );
```

我们创建了包含了事件名称（`["stock","stock-update"]`）的数组，然后 `map(..)`（见第 8 章）这个数组，生成了一个包含了两个 observable 的数组，然后把这个数组和 observable 映射函数 `zip(..)`（见第 8 章）起来，产生一个 `[ observable, mapperFn ]` 这样的元组数组。最后通过 `spreadArgs(..)`（见第 3 章）把每个元组数组展开为单独的参数，`map(..)` 到了 `transformObservable(..)` 函数上。

得到的结果是一个包含了转换好的 observable 的数组，通过数组结构赋值的方式分别赋值到了 `newStocks` 和 `stockUpdates` 两个变量上。

到此为止，我们用轻量级函数式编程的方式来让股票行情信息事件成为了 observable！在 `ch11-code/stock-ticker.js` 中我们会订阅这两个 observable。

回头想想我们用到的函数式编程原则。这样做有没有意义呢？你能否明白我们是如何运用前几章中介绍的各种概念的呢？你能不能想到别的方式来实现这些功能？

更重要的是，如果你用命令式编程的方法是如何实现上面的功能的呢？你认为两种方式相比孰优孰劣？试试看用你熟悉的命令式编程的方式去写这个功能。如果你和我一样，那么命令式编程仍然会让你感到更加自然。

在进行下面的学习之前，你需要明白的是，除了使你感到非常自然的命令式编程以外，你也已经能够了解函数式编程的合理性了。想想看每个函数的输入和输出，你看到它们是怎样组合在一起的了吗？

在你豁然开朗以前一定要持续不断地练习。

股票行情界面

如果你熟悉了上一章节中的函数式编程模式，你就可以开始学习 `ch11-code/stock-ticker.js` 文件中的内容了。这里会涉及相当多的重要内容，所以我们将好好地理解整个文件中的每个方法。

我们先从定义一些操作 DOM 的辅助函数开始：

```
function isTextNode(node) {
    return node && node.nodeType == 3;
}
function getElemAttr(elem,prop) {
    return elem.getAttribute( prop );
}
function setElemAttr(elem,prop,val) {
    // 副作用！！
    return elem.setAttribute( prop, val );
}
function matchingStockId(id) {
    return function isStock(node){
        return getStockId( node ) == id;
    };
}
function isStockInfoChildElem(elem) {
    return /\bstock-/i.test( getClassName( elem ) );
}
function appendDOMChild(parentNode,childNode) {
    // 副作用！！
    parentNode.appendChild( childNode );
    return parentNode;
}
function setDOMContent(elem,html) {
    // 副作用！！
    elem.innerHTML = html;
    return elem;
}

var createElement = document.createElement.bind( document );

var getElemAttrByName = curry( reverseArgs( getElemAttr ), 2 );
var getStockId = getElemAttrByName( "data-stock-id" );
var getClassName = getElemAttrByName( "class" );
```

这些函数应该算是不言自明的。为了获得 `getElemAttrByName(..)`，我用了 `curry(reverseArgs( .. ))`（见第 3 章）而不是 `partialRight(..)`，只是为了在这种特殊情况下，稍微提高一点性能。

注意，我标出了操作 DOM 元素时的副作用。因为不能简单地用克隆的 DOM 对象去替换已有的，所以我们在不替换已有对象的基础上，勉强接受了一些副作用的产生。至少如果在 DOM 渲染中产生一个错误，我们可以轻松地搜索这些代码注释来缩小可能的错误代码。

`matchingStockId(..)` 用到了闭包（见第 2 章），它创建了一个内部函数（`isStock(..)`），使在其他作用域下运行时依然能够保存 `id` 变量。

其他的辅助函数：

```
function stripPrefix(prefixRegex) {
    return function mapperFn(val) {
        return val.replace( prefixRegex, "" );
    };
}
function listify(listOrItem) {
    if (!Array.isArray( listOrItem )) {
        return [ listOrItem ];
    }
    return listOrItem;
}
```

定义一个用以获取某个 DOM 元素的子节点的辅助函数：

```
var getDOMChildren = pipe(
    listify,
    flatMap(
        pipe(
            curry( prop )( "childNodes" ),
            Array.from
        )
    )
);
```

首先，用 `listify(..)` 来保证我们得到的是一个数组（即使里面只有一个元素）。回忆一下在第 8 章中提到的 `flatMap(..)`，这个函数把一个包含数组的数组扁平化，变成一个浅数组。

映射函数先把 DOM 元素映射成它的子元素数组，然后我们用 `Array.from(..)` 把这个数组变成一个真实的数组（而不是一个 NodeList）。这两个函数组合成一个映射函数（通过 `pipe(..)`），这就是融合（见第 8 章）。

现在，我们用 `getDOMChildren(..)` 实用函数来定义股票行情工具中查找特定 DOM 元素的工具函数：

```
function getStockElem(tickerElem,stockId) {
    return pipe(
        getDOMChildren,
        filterOut( isTextNode ),
        filterIn( matchingStockId( stockId ) )
    )
    ( tickerElem );
}
function getStockInfoChildElems(stockElem) {
    return pipe(
        getDOMChildren,
        filterOut( isTextNode ),
        filterIn( isStockInfoChildElem )
    )
    ( stockElem );
}
```

`getStockElem(..)` 接受 `tickerElem` DOM 节点作为参数，获取其子元素，然后过滤，保证我们得到的是符合股票代码的 DOM 元素。 `getStockInfoChildElems(..)` 几乎是一样的，不同的是它从一个股票元素节点开始查找，还使用了不同的过滤函数。

两个实用函数都会过滤掉文字节点（因为它们没有其他的 DOM 节点那样的方法），保证返回一个 DOM 元素数组，哪怕数组中只有一个元素。

主函数

我们用 `stockTickerUI` 对象来保存三个修改界面的主要方法，如下：

```
var stockTickerUI = {

    updateStockElems(stockInfoChildElemList,data) {
        // ..
    },

    updateStock(tickerElem,data) {
        // ..
    },

    addStock(tickerElem,data) {
        // ..
    }
};
```

我们先看看 `updateStock(..)` ，这是三个函数里面最简单的：

```
var stockTickerUI = {

    // ..

    updateStock(tickerElem,data) {
        var getStockElemFromId = curry( getStockElem )( tickerElem );
        var stockInfoChildElemList = pipe(
            getStockElemFromId,
            getStockInfoChildElems
        )
        ( data.id );

        return stockTickerUI.updateStockElems(
            stockInfoChildElemList,
            data
        );
    },

    // ..

};
```

柯里化之前的辅助函数 `getStockElem(..)`，传给它 `tickerElem`，得到了 `getStockElemFromId(..)` 函数，这个函数接受 `data.id` 作为参数。把 `<li>` 元素（其实是数组形式的）传入 `getStockInfoChildElems(..)`，我们得到了三个 `<span>` 子元素，用来展示股票信息，我们把它们保存在 `stockInfoChildElemList` 变量中。然后把数组和股票信息 `data` 对象一起传给 `stockTickerUI.updateStockElems(..)`，来更新 `<span>` 中的数据。

现在我们来看看 `stockTickerUI.updateStockElems(..)`：

```
var stockTickerUI = {

    updateStockElems(stockInfoChildElemList,data) {
        var getDataVal = curry( reverseArgs( prop ), 2 )( data );
        var extractInfoChildElemVal = pipe(
            getClassName,
            stripPrefix( /\bstock-/i ),
            getDataVal
        );
        var orderedDataVals =
            map( extractInfoChildElemVal )( stockInfoChildElemList );
        var elemsValsTuples =
            filterOut( function updateValueMissing([infoChildElem,val]){
                return val === undefined;
            } )
            ( zip( stockInfoChildElemList, orderedDataVals ) );

        // 副作用！！
        compose( each, spreadArgs )
        ( setDOMContent )
        ( elemsValsTuples );
    },

    // ..

};
```

这部分有点难理解。我们一行行来看。

首先把 `prop` 函数的参数反转，柯里化后，把 `data` 消息对象绑定上去，得到了 `getDataVal(..)` 函数，这个函数接收一个属性名称作为参数，返回 `data` 中的对应的属性名称的值。

接下来，我们看看 `extractInfoChildElem`：

```
var extractInfoChildElemVal = pipe(
    getClassName,
    stripPrefix( /\bstock-/i ),
    getDataVal
);
```

这个函数接受一个 DOM 元素作为参数，拿到 class 属性的值，然后把 `"stock-"` 前缀去掉，然后用这个属性值（`"name"`，`"price"` 或 `"change"`），通过 `getDataVal(..)` 函数，在 `data` 中找到对应的数据。你可能会问："还有这种操作？"。

其实，这么做的目的是按照 `stockInfoChildElemList` 中的 `<span>` 元素的顺序从 `data` 中拿到数据。我们对 `stockInfoChildElemList` 数组调用 `extractInfoChildElem` 映射函数，来拿到这些数据。

接下来，我们把 `<span>` 数组和数据数组压缩起来，得到一个元组：

```
zip( stockInfoChildElemList, orderedDataVals )
```

这里有一点不太容易理解，我们定义的 observable 转换函数中，新的股票行情数据 `data` 会包含一个 `name` 属性，来对应 `<span class="stock-name">` 元素，但是在股票行情更新事件的数据中可能会找不到对应的 `name` 属性。

一般来说，如果股票更新消息事件的数据对象不包含某个股票数据的话，我们就不应该更新这只股票对应的 DOM 元素。所以我们要用 `filterOut(..)` 剔除掉没有值的元组（这里的值在元组的第二个元素）。

```
var elemsValsTuples =
    filterOut( function updateValueMissing([infoChildElem,val]){
        return val === undefined;
    } )
    ( zip( stockInfoChildElemList, orderedDataVals ) );
```

筛选后的结果是一个元组数组（如：`[ <span>, ".." ]`），这个数组可以用来更新 DOM 了，我们把这个结果保存到 `elemsValsTuples` 变量中。

注意： 既然 `updateValueMissing(..)` 是声明在函数内的，所以我们可以更方便地控制这个函数。与其使用 `spreadArgs(..)` 来把函数接收的一个数组形式的参数展开成两个参数，我们可以直接用函数的参数解构声明（`function updateValueMissing([infoChildElem,val]){ ..`），参见第 2 章。

最后，我们要更新 DOM 中的 `<span>` 元素：

```
// 副作用！！
compose( each, spreadArgs )( setDOMContent )
( elemsValsTuples );
```

我们用 `each(..)` 遍历了 `elemsValsTuples` 数组（参考第 8 章中关于 `forEach(..)` 的讨论）。

与其他地方使用 `pipe(..)` 来组合函数不同，这里使用 `compose(..)`（见第 4 章），先把 `setDomContent(..)` 传到 `spreadArgs(..)` 中，再把执行的结果作为迭代函数传到 `each(..)` 中。执行时，每个元组被展开为参数传给了 `setDOMContent(..)` 函数，然后对应地更新 DOM 元素。

最后说明下 `addStock(..)`。我们先把整个函数写出来，然后再一句句地解释：

```
var stockTickerUI = {

    // ..

    addStock(tickerElem,data) {
        var [stockElem, ...infoChildElems] = map(
            createElement
        )
        ( [ "li", "span", "span", "span" ] );
        var attrValTuples = [
            [ ["class","stock"], ["data-stock-id",data.id] ],
            [ ["class","stock-name"] ],
            [ ["class","stock-price"] ],
            [ ["class","stock-change"] ]
        ];
        var elemsAttrsTuples =
            zip( [stockElem, ...infoChildElems], attrValTuples );

        // 副作用！！
        each( function setElemAttrs([elem,attrValTupleList]){
            each(
                spreadArgs( partial( setElemAttr, elem ) )
            )
            ( attrValTupleList );
        } )
        ( elemsAttrsTuples );

        // 副作用！！
        stockTickerUI.updateStockElems( infoChildElems, data );
        reduce( appendDOMChild )( stockElem )( infoChildElems );
        tickerElem.appendChild( stockElem );
    }

};
```

这个操作界面的函数会根据新的股票信息生成一个空的 DOM 结构，然后调用 `stockTickerUI.updateStockElems(..)` 方法来更新其中的内容。

首先：

```
var [stockElem, ...infoChildElems] = map(
    createElement
)
( [ "li", "span", "span", "span" ] );
```

我们先创建 `<li>` 父元素和三个 `<span>` 子元素，把它们分别赋值给了 `stockElem` 和 `infoChildElems` 数组。

为了设置 DOM 元素的对应属性，我们声明了一个元组数组组成的数组。按照顺序，每个元组数组对应上面四个 DOM 元素中的一个。每个元组数组中的元组由对应元素的属性和值组成：

```
var attrValTuples = [
    [ ["class","stock"], ["data-stock-id",data.id] ],
    [ ["class","stock-name"] ],
    [ ["class","stock-price"] ],
    [ ["class","stock-change"] ]
];
```

我们把四个 DOM 元素和 `attrValTuples` 数组 `zip(..)` 起来：

```
var elemsAttrsTuples =
    zip( [stockElem, ...infoChildElems], attrValTuples );
```

最后的结果会是：

```
[
    [ <li>, [ ["class","stock"], ["data-stock-id",data.id] ] ],
    [ <span>, [ ["class","stock-name"] ] ],
    ..
]
```

如果我们用命令式的方式来把属性和值设置到每个 DOM 元素上，我们会用嵌套的 `for` 循环。用函数式编程的方式的话也会是这样，不过这时嵌套的是 `each(..)` 循环：

```
// 副作用！！
each( function setElemAttrs([elem,attrValTupleList]){
    each(
        spreadArgs( partial( setElemAttr, elem ) )
    )
    ( attrValTupleList );
} )
( elemsAttrsTuples );
```

外层的 `each(..)` 循环了元组数组，其中每个数组的元素是一个 `elem` 和它对应的 `attrValTupleList`，这个元组数组被传入了 `setElemAttrs(..)`，在函数的参数中被解构成两个值。

在外层循环内，元组数组的子数组（包含了属性和值的数组）被传递到了内层的 `each(..)` 循环中。内层的迭代函数首先以 `elem` 作为第一个参数对 `setElemAttr(..)` 进行了部分实现，然后把剩下的函数参数展开，把每个属性值元组作为参数传递进这个函数中。

到此为止，我们有了 `<span>` 元素数组，每个元素上都有了该有的属性，但是还没有 `innerHTML` 的内容。这里，我们要用 `stockTickerUI.updateStockElems(..)` 函数，把 `data` 设置到 `<span>` 上去，和股票信息更新事件的处理一样。

然后，我们要把这些 `<span>` 元素添加到对应的父级 `<li>` 元素中去，我们用 `reduce(..)` 来做这件事（见第 8 章）。

```
reduce( appendDOMChild )( stockElem )( infoChildElems );
```

最后，用操作 DOM 元素的副作用方法把新的股票元素添加到小工具的 DOM 节点中去：

```
tickerElem.appendChild( stockElem );
```

呼！你跟上了吗？我建议你在继续下去之前，回到开头，重新读几遍这部分内容，再练习几遍。

订阅 Observable

最后一个重要任务是订阅 `ch11-code/stock-ticker-events.js` 中定义的 observable，把事件传递给正确的主函数（`addStock(..)` 和 `updateStock(..)`）。

注意，这两个主函数接受 `tickerElem` 作为第一个参数。我们声明一个数组（`stockTickerUIMethodsWithDOMContext`）保存了两个中间函数（也叫作闭包，见第 2 章），这两个中间函数是通过部分参数绑定的函数把小工具的 DOM 元素绑定到了两个主函数上来生成的。

```
var ticker = document.getElementById( "stock-ticker" );

var stockTickerUIMethodsWithDOMContext = map(
    curry( reverseArgs( partial ), 2 )( ticker )
)
( [ stockTickerUI.addStock, stockTickerUI.updateStock ] );
```

`reverseArgs( partial )` 是之前提到的 `partialRight(..)` 的替代品，优化了性能。但是这里 `partial(..)` 是映射函数的目标函数。所以我们需要事先 `curry(..)` 化，这样我们就可以先把第二个参数 `ticker` 传给 `partial(..)`，后面把主函数传进去的时候就可以用到之前传入的 `ticker` 了。数组中的这两个中间函数就可以被用来订阅 observable 了。

我们用闭包在这两个中间函数中保存了 `ticker` 数据，在第 7 章中，我们知道了还可以把 `ticker` 保存在对象的属性上，通过使用两个函数上的指向 `stockTickerUI` 的 `this` 来访问 `ticker`。因为 `this` 是个隐式的输入（见第 2 章），所以一般来说不推荐用对象的方式，所以我使用了闭包的方式。

为了订阅 observable，我们先写一个辅助函数，提供一个未绑定的方法：

```
var subscribeToObservable =
    pipe( uncurry, spreadArgs )( unboundMethod( "subscribe" ) );
```

`unboundMethod("subscribe")` 已经柯里化了，所以我们用 `uncurry(..)`（见第 3 章）先反柯里化，然后再用 `spreadArgs(..)`（依然见第 3 章）来修改接受的参数的格式，所以这个函数接受一个元组作为参数，展开后传递下去。

现在，我们只要把 observable 数组和封装好上下文的主函数 `zip(..)` 起来。生成一个元组数组，每个元组可以用之前定义的 `subscribeToObservable(..)` 辅助函数来订阅 observable：

```
var stockTickerObservables = [ newStocks, stockUpdates ];

// 副作用！！
each( subscribeToObservable )
( zip( stockTickerUIMethodsWithDOMContext, stockTickerObservables ) );
```

由于我们修改了这些 observable 的状态以订阅它们，而且由于我们使用了 `each(..)` —— 总是和副作用相关！ —— 我们用代码注释来说明这个问题。

就是这样！花些时间研究比较这段代码和它命令式的替代版本，正如我们之前在股票行情信息中讨论到的一样。真的，可以多花点时间。我知道这是一本很长的书，但是完整地读下来会让你能够消化和理解这样的代码。

你现在打算在 JavaScript 中如何合理地使用函数式编程？继续练习，就像我们在这里做的一样！

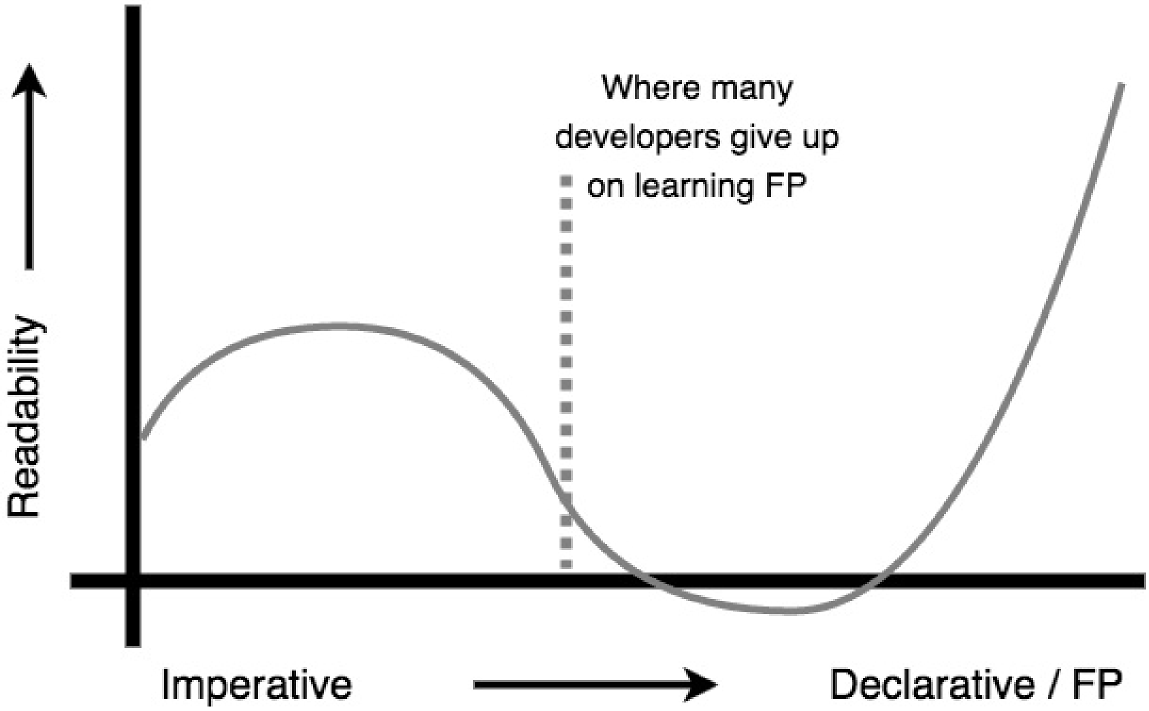
总结

我们在本章中讨论的示例代码应该被作为一个整体来阅读，而不仅仅是作为章节中所展示的支离破碎的代码片段。如果你还没有完整地阅读过，现在请停下来，去完整地阅读一遍代码目录下的文件吧。确保你在完整的上下文中了解它们。

示例代码并不是实际编写代码的范例，只是提供了一种描述性的，教授如何用轻量级函数式的技巧来解决此类问题的方法。这些代码尽可能多地把本书中不同概念联系起来。这里提供了比代码片段更真实的例子来学习函数式编程。

我相信，随着我不断地学习函数式编程，我会继续改进这个示例代码。你现在看到的只是我在学习曲线上的一个快照。我希望对你来说也是如此。

在我们结束本书的主要内容时，我们一起回顾一下我在第 1 章中提到的可读性曲线：



在学习函数式编程的过程中，理解这张图的真谛，并且为自己设定合理的预期，是非常重要的。你已经到这里了，这已经是一个很大的成果了。

但是，当你在绝望和沮丧的低谷时，别停下来。前面等待你的是一种更好的思维方式，可以写出可读性更好，更容易理解，更容易验证，最终更加可靠的代码。

我不需要再为开发者们不断前行想出更多崇高的理由。感谢你参与到我学习 JavaScript 中的函数式编程的原理的过程中来。我希望你的学习过程和我的一样，充实而充满希望！

JavaScript 轻量级函数式编程

附录 A：Transducing

Transducing 是我们这本书要讲到的更为高级的技术。它继承了第 8 章数组操作的许多思想。

我不会把 Transducing 严格的称为“轻量级函数式编程”，它更像是一个顶级的技巧。我把这个技术留到附录来讲意味着你现在很可能并不需要关心它，当你确保你已经非常熟悉整本书的主要内容，你可以再回头看看这一章节。

说实话，即使我已经教过 transducing 很多次了，在写这一章的时候，我仍然需要花很多脑力去理清楚这个技术。所以，如果你看这一章看的很疑惑也没必要感到沮丧。把这一章加个书签，等你觉得你差不多能理解时再回头看看。

Transducing 就是通过减少来转换。

我知道这听起来很令人费解。但是让我们来看看它有多强大。实际上，我认为这是你掌握了轻量级函数式编程后可以做的最好的例证之一。

和这本书的其他部分一样，我的方法是先解释为什么使用这个技术，然后如何使用，最后归结为简单的这个技术到底是什么样的。这通常会有多学很多东西，但是我觉得用这种方式你会更深入的理解它。

首先，为什么

让我们从扩展我们在第 3 章中介绍的例子开始，测试单词是否足够短和/或足够长：

```
function isLongEnough(str) {
    return str.length >= 5;
}

function isShortEnough(str) {
    return str.length <= 10;
}
```

在第 3 章中，我们使用这些断言函数来测试一个单词。然后在第 8 章中，我们学习了如何使用像 `filter(..)` 这样的数组操作来重复这些测试。例如:

```
var words = [ "You", "have", "written", "something", "very", "interesting" ];

words
.filter( isLongEnough )
.filter( isShortEnough );
// ["written","something"]
```

这个例子可能并不明显，但是这种分开操作相同数组的方式具有一些不理想的地方。当我们处理一个值比较少的数组时一切都还好。但是如果数组中有很多值，每个 `filter(..)` 分别处理数组的每个值会比我们预期的慢一点。

当我们的数组是异步/懒惰（也称为 observables）的，随着时间的推移响应事件处理（见第 10 章），会出现类似的性能问题。在这种情况下，一次事件只有一个值，因此使用两个单独的 `filter(..)` 函数处理这些值并不是什么大不了的事情。

但是，不太明显的是每个 `filter(..)` 方法都会产生一个单独的 observable 值。从一个 observable 值中抽出一个值的开销真的可以加起来（译者注：详情请看第 10 章的“积极的 vs 惰性的”这一节）。这是真实存在的，因为在这些情况下，处理数千或数百万的值并不罕见; 所以，即使是这么小的成本也会很快累加起来。

另一个缺点是可读性，特别是当我们需要对多个数组（或 observable）重复相同的操作时。例如:

```
zip(
    list1.filter( isLongEnough ).filter( isShortEnough ),
    list2.filter( isLongEnough ).filter( isShortEnough ),
    list3.filter( isLongEnough ).filter( isShortEnough )
)
```

显得很重复，对不对？

如果我们可以将 `isLongEnough(..)` 断言与 `isShortEnough(..)` 断言组合在一起是不是会更好一点呢（可读性和性能）？你可以手动执行：

```
function isCorrectLength(str) {
    return isLongEnough( str ) && isShortEnough( str );
}
```

但这不是函数式编程的方式！

在第 8 章中，我们讨论了融合 —— 组合相邻映射函数。回忆一下：

```
words
.map(
    pipe( removeInvalidChars, upper, elide )
);
```

不幸的是，组合相邻断言函数并不像组合相邻映射函数那样容易。为什么呢？想想断言函数长什么“样子”——一种描述输入和输出的学术方式。它接收一个单一的参数，返回一个 true 或 false。

如果你试着用 `isshortenough(islongenough(str))`，这是行不通的。因为 `islongenough(..)` 会返回 true 或者 false，而不是返回 `isshortenough(..)` 所要的字符串类型的值。这可真倒霉。

试图组合两个相邻的 reducer 函数同样是行不通的。reducer 函数接收两个值作为输入，并返回单个组合值。reducer 函数的单一返回值也不能作为参数传到另一个需要两个输入的 reducer 函数中。

此外，`reduce(..)` 辅助函数可以接收一个可选的 `initialValue` 输入。有时可以省略，但有时候它又必须被传入。这就让组合更复杂了，因为一个 `reduce(..)` 可能需要一个 `initialValue`，而另一个 `reduce(..)` 可能需要另一个 `initialValue`。所以我们怎么可能只用某种组合的 reducer 来实现 `reduce(..)` 呢。

考虑像这样的链：

```
words
.map( strUppercase )
.filter( isLongEnough )
.filter( isShortEnough )
.reduce( strConcat, "" );
// "WRITTENSOMETHING"
```

你能想出一个组合能够包含 `map(strUppercase)`，`filter(isLongEnough)`，`filter(isShortEnough)`，`reduce(strConcat)` 所有这些操作吗？每种操作的行为是不同的，所以不能直接组合在一起。我们需要把它们修改下让它们组合在一起。

希望这些例子说明了为什么简单的组合不能胜任这项任务。我们需要一个更强大的技术，而 transducing 就是这个技术。

如何，下一步

让我们谈谈我们该如何得到一个能组合映射，断言和/或 reducers 的框架。

别太紧张：你不必经历编程过程中所有的探索步骤。一旦你理解了 transducing 能解决的问题，你就可以直接使用函数式编程库中的 `transduce(..)` 工具继续你应用程序的剩余部分！

让我们开始探索吧。

把 Map/Filter 表示为 Reduce

我们要做的第一件事情就是将我们的 `filter(..)` 和 `map(..)` 调用变为 `reduce(..)` 调用。回想一下我们在第 8 章是怎么做的：

```
function strUppercase(str) { return str.toUpperCase(); }
function strConcat(str1,str2) { return str1 + str2; }

function strUppercaseReducer(list,str) {
    list.push( strUppercase( str ) );
    return list;
}

function isLongEnoughReducer(list,str) {
    if (isLongEnough( str )) list.push( str );
    return list;
}

function isShortEnoughReducer(list,str) {
    if (isShortEnough( str )) list.push( str );
    return list;
}

words
.reduce( strUppercaseReducer, [] )
.reduce( isLongEnoughReducer, [] )
.reduce( isShortEnough, [] )
.reduce( strConcat, "" );
// "WRITTENSOMETHING"
```

这是一个不错的改进。我们现在有四个相邻的 `reduce(..)` 调用，而不是三种不同方法的混合。然而，我们仍然不能 `compose(..)` 这四个 reducer，因为它们接受两个参数而不是一个参数。

在 8 章，我们偷了点懒使用了数组的 `push` 方法而不是 `concat(..)` 方法返回一个新数组，导致有副作用。现在让我们更正式一点：

```
function strUppercaseReducer(list,str) {
    return list.concat( [strUppercase( str )] );
}

function isLongEnoughReducer(list,str) {
    if (isLongEnough( str )) return list.concat( [str] );
    return list;
}

function isShortEnoughReducer(list,str) {
    if (isShortEnough( str )) return list.concat( [str] );
    return list;
}
```

在后面我们会来头看看这里是否需要 `concat(..)` 。

参数化 Reducers

除了使用不同的断言函数之外，两个 filter reducers 几乎相同。让我们把这些 reducers 参数化得到一个可以定义任何 filter-reducer 的工具函数：

```
function filterReducer(predicateFn) {
    return function reducer(list,val){
        if (predicateFn( val )) return list.concat( [val] );
        return list;
    };
}

var isLongEnoughReducer = filterReducer( isLongEnough );
var isShortEnoughReducer = filterReducer( isShortEnough );
```

同样的，我们把 `mapperFn(..)` 也参数化来生成 map-reducer 函数：

```
function mapReducer(mapperFn) {
    return function reducer(list,val){
        return list.concat( [mapperFn( val )] );
    };
}

var strToUppercaseReducer = mapReducer( strUppercase );
```

我们的调用链看起来是一样的：

```
words
.reduce( strUppercaseReducer, [] )
.reduce( isLongEnoughReducer, [] )
.reduce( isShortEnough, [] )
.reduce( strConcat, "" );
```

提取共用组合逻辑

仔细观察上面的 `mapReducer(..)` 和 `filterReducer(..)` 函数。你发现共享功能了吗？

这部分：

```
return list.concat( .. );

// 或者
return list;
```

让我们为这个通用逻辑定义一个辅助函数。但是我们叫它什么呢？

```
function WHATSITCALLED(list,val) {
    return list.concat( [val] );
}
```

`WHATSITCALLED(..)` 函数做了些什么呢，它接收两个参数（一个数组和另一个值），将值 concat 到数组的末尾返回一个新的数组。所以这个 `WHATSITCALLED(..)` 名字不合适，我们可以叫它 `listCombination(..)` ：

```
function listCombination(list,val) {
    return list.concat( [val] );
}
```

我们现在用 `listCombination(..)` 来重新定义我们的 reducer 辅助函数：

```
function mapReducer(mapperFn) {
    return function reducer(list,val){
        return listCombination( list, mapperFn( val ) );
    };
}

function filterReducer(predicateFn) {
    return function reducer(list,val){
        if (predicateFn( val )) return listCombination( list, val );
        return list;
    };
}
```

我们的调用链看起来还是一样的（这里就不重复写了）。

参数化组合

我们的 `listCombination(..)` 小工具只是组合两个值的一种方式。让我们将它的用途参数化，以使我们的 reducers 更加通用：

```
function mapReducer(mapperFn,combinationFn) {
    return function reducer(list,val){
        return combinationFn( list, mapperFn( val ) );
    };
}

function filterReducer(predicateFn,combinationFn) {
    return function reducer(list,val){
        if (predicateFn( val )) return combinationFn( list, val );
        return list;
    };
}
```

使用这种形式的辅助函数：

```
var strToUppercaseReducer = mapReducer( strUppercase, listCombination );
var isLongEnoughReducer = filterReducer( isLongEnough, listCombination );
var isShortEnoughReducer = filterReducer( isShortEnough, listCombination );
```

将这些实用函数定义为接收两个参数而不是一个参数不太方便组合，因此我们使用我们的 `curry(..)` （柯里化）方法：

```
var curriedMapReducer = curry( function mapReducer(mapperFn,combinationFn){
    return function reducer(list,val){
        return combinationFn( list, mapperFn( val ) );
    };
} );

var curriedFilterReducer = curry( function filterReducer(predicateFn,combinationFn){
    return function reducer(list,val){
        if (predicateFn( val )) return combinationFn( list, val );
        return list;
    };
} );

var strToUppercaseReducer =
    curriedMapReducer( strUppercase )( listCombination );
var isLongEnoughReducer =
    curriedFilterReducer( isLongEnough )( listCombination );
var isShortEnoughReducer =
    curriedFilterReducer( isShortEnough )( listCombination );
```

这看起来有点冗长而且可能不是很有用。

但这实际上是我们进行下一步推导的必要条件。请记住，我们的最终目标是能够 `compose(..)` 这些 reducers。我们快要完成了。

组合柯里化

这一步是最棘手的。所以请慢慢的用心的阅读。

让我们看看没有将 `listCombination(..)` 传递给柯里化函数的样子：

```
var x = curriedMapReducer( strUppercase );
var y = curriedFilterReducer( isLongEnough );
var z = curriedFilterReducer( isShortEnough );
```

看看这三个中间函数 `x(..)`，`y(..)` 和 `z(..)`。每个函数都期望得到一个单一的组合函数并产生一个 reducer 函数。

记住，如果我们想要所有这些的独立的 reducer，我们可以这样做：

```
var upperReducer = x( listCombination );
var longEnoughReducer = y( listCombination );
var shortEnoughReducer = z( listCombination );
```

但是，如果你调用 `y(z)`，会得到什么呢？当把 `z` 传递给 `y(..)` 调用，而不是 `combinationFn(..)` 时会发生什么呢？这个返回的 reducer 函数内部看起来像这样：

```
function reducer(list,val) {
    if (isLongEnough( val )) return z( list, val );
    return list;
}
```

看到 `z(..)` 里面的调用了吗？这看起来应该是错误的，因为 `z(..)` 函数应该只接收一个参数（ `combinationFn(..)` ），而不是两个参数（list 和 val）。这和要求不匹配。不行。

我们来看看组合 `y(z(listCombination))` 。我们将把它分成两个不同的步骤：

```
var shortEnoughReducer = z( listCombination );
var longAndShortEnoughReducer = y( shortEnoughReducer );
```

我们创建 `shortEnoughReducer(..)` ，然后将它作为 `combinationFn(..)` 传递给 `y(..)` ，生成 `longAndShortEnoughReducer(..)` 。多读几遍，直到理解。

现在想想： `shortEnoughReducer(..)` 和 `longAndShortEnoughReducer(..)` 的内部构造是什么样的呢？你能想得到吗？

```
// shortEnoughReducer, from z(..):
function reducer(list,val) {
    if (isShortEnough( val )) return listCombination( list, val );
    return list;
}

// longAndShortEnoughReducer, from y(..):
function reducer(list,val) {
    if (isLongEnough( val )) return shortEnoughReducer( list, val );
    return list;
}
```

你看到 `shortEnoughReducer(..)` 替代了 `longAndShortEnoughReducer(..)` 里面 `listCombination(..)` 的位置了吗？为什么这样也能运行？

因为 **`reducer(..)`** 的“形状”和 **`listCombination(..)`** 的形状是一样的。换句话说，reducer 可以用作另一个 reducer 的组合函数; 它们就是这样组合起来的！ `listCombination(..)` 函数作为第一个 reducer 的组合函数，这个 reducer 又可以作为组合函数给下一个 reducer，以此类推。

我们用几个不同的值来测试我们的 `longAndShortEnoughReducer(..)` ：

```
longAndShortEnoughReducer( [], "nope" );
// []

longAndShortEnoughReducer( [], "hello" );
// ["hello"]

longAndShortEnoughReducer( [], "hello world" );
// []
```

`longAndShortEnoughReducer(..)` 会过滤出不够长且不够短的值，它在同一步骤中执行这两个过滤。这是一个组合 reducer！

再花点时间消化下。

现在，把 `x(..)` （生成大写 reducer 的产生器）加入组合：

```
var longAndShortEnoughReducer = y( z( listCombination) );
var upperLongAndShortEnoughReducer = x( longAndShortEnoughReducer );
```

正如 `upperLongAndShortEnoughReducer(..)` 名字所示，它同时执行所有三个步骤 - 一个映射和两个过滤器！它内部看起来是这样的：

```
// upperLongAndShortEnoughReducer:
function reducer(list,val) {
    return longAndShortEnoughReducer( list, strUppercase( val ) );
}
```

一个字符串类型的 `val` 被传入，由 `strUppercase(..)` 转换成大写，然后传递给 `longAndShortEnoughReducer(..)`。该函数只有在 `val` 满足足够长且足够短的条件时才将它添加到数组中。否则数组保持不变。

我花了几个星期来思考分析这种杂耍似的操作。所以别着急，如果你需要在这好好研究下，重新阅读个几（十几个）次。慢慢来。

现在来验证一下：

```
upperLongAndShortEnoughReducer( [], "nope" );
// []

upperLongAndShortEnoughReducer( [], "hello" );
// ["HELLO"]

upperLongAndShortEnoughReducer( [], "hello world" );
// []
```

这个 reducer 成功的组合了和 map 和两个 filter，太棒了！

让我们回顾一下我们到目前为止所做的事情：

```
var x = curriedMapReducer( strUppercase );
var y = curriedFilterReducer( isLongEnough );
var z = curriedFilterReducer( isShortEnough );

var upperLongAndShortEnoughReducer = x( y( z( listCombination ) ) );

words.reduce( upperLongAndShortEnoughReducer, [] );
// ["WRITTEN","SOMETHING"]
```

这已经很酷了，但是我们可以让它更好。

`x(y(z( .. )))` 是一个组合。我们可以直接跳过中间的 `x` / `y` / `z` 变量名，直接这么表示该组合：

```
var composition = compose(
    curriedMapReducer( strUppercase ),
    curriedFilterReducer( isLongEnough ),
    curriedFilterReducer( isShortEnough )
);

var upperLongAndShortEnoughReducer = composition( listCombination );

words.reduce( upperLongAndShortEnoughReducer, [] );
// ["WRITTEN","SOMETHING"]
```

我们来考虑下该组合函数中“数据”的流动：

1. `listCombination(..)` 作为组合函数传入，构造 `isShortEnough(..)` 过滤器的 reducer。
2. 然后，所得到的 reducer 函数作为组合函数传入，继续构造 `isShortEnough(..)` 过滤器的 reducer。
3. 最后，所得到的 reducer 函数作为组合函数传入，构造 `strUppercase(..)` 映射的 reducer。

在前面的片段中， `composition(..)` 是一个组合函数，期望组合函数来形成一个 reducer；而这个 `composition(..)` 有一个特殊的标签：transducer。给 transducer 提供组合函数产生组合的 reducer：

// TODO：检查 transducer 是产生 reducer 还是它本身就是 reducer

```
var transducer = compose(
    curriedMapReducer( strUppercase ),
    curriedFilterReducer( isLongEnough ),
    curriedFilterReducer( isShortEnough )
);

words
.reduce( transducer( listCombination ), [] );
// ["WRITTEN","SOMETHING"]
```

注意：我们应该好好观察下前面两个片段中的 `compose(..)` 顺序，这地方有点难理解。回想一下，在我们的原始示例中，我们先 `map(strUppercase)` 然后 `filter(isLongEnough)` ，最后 `filter(isShortEnough)` ；这些操作实际上也确实按照这个顺序执行的。但在第 4 章中，我们了解到， `compose(..)` 通常是以相反的顺序运行。那么为什么我们不需要反转这里的顺序来获得同样的期望结果呢？来自每个 reducer 的 `combinationFn(..)` 的抽象反转了操作顺序。所以和直觉相反，当组合一个 tranducer 时，你只需要按照实际的顺序组合就好！

列表组合：纯与不纯

我们再来看一下我们的 `listCombination(..)` 组合函数的实现：

```
function listCombination(list,val) {
    return list.concat( [val] );
}
```

虽然这种方法是纯的，但它对性能有负面影响。首先，它创建临时数组来包裹 `val` 。然后， `concat(..)` 方法创建一个全新的数组来连接这个临时数组。每一步都会创建和销毁的很多数组，这不仅对 CPU 不利，也会造成 GC 内存的流失。

下面是性能更好但是不纯的版本：

```
function listCombination(list,val) {
    list.push( val );
    return list;
}
```

单独的考虑下 `listCombination(..)` ，毫无疑问，这是不纯的，这通常是我们想要避免的。但是，我们应该考虑一个更大的背景。

`listCombination(..)` 不是我们完全有交互的函数。我们不直接在程序中的任何地方使用它，而只是在 transducing 的过程中使用它。

回到第 5 章，我们定义纯函数来减少副作用的目标只是限制在应用的 API 层级。对于底层实现，只要没有违反对外部是纯函数，就可以在函数内为了性能而变得不纯。

`listCombination(..)` 更多的是转换的内部实现细节。实际上，它通常由 transducing 库提供！而不是你的程序中进行交互的顶层方法。

底线：我认为甚至使用 `listCombination(..)` 的性能最优但是不纯的版本也是完全可以接受的。只要确保你用代码注释记录下它不纯即可！

可选的组合

到目前为止，这是我们用转换所得到的：

```
words
.reduce( transducer( listCombination ), [] )
.reduce( strConcat, "" );
// 写点什么
```

这已经非常棒了，但是我们还藏着最后一个的技巧。坦白来说，我认为这部分能够让你迄今为止付出的所有努力变得值得。

我们可以用某种方式实现只用一个 `reduce(..)` 来“组合”这两个 `reduce(..)` 吗？ 不幸的是，我们并不能将 `strConcat(..)` 添加到 `compose(..)` 调用中; 它的“形状”不适用于那个组合。

但是让我们来看下这两个功能：

```
function strConcat(str1,str2) { return str1 + str2; }

function listCombination(list,val) { list.push( val ); return list; }
```

如果你用心观察，可以看出这两个功能是如何互换的。它们以不同的数据类型运行，但在概念上它们也是一样的：将两个值组合成一个。

换句话说， `strConcat(..)` 是一个组合函数！

这意味着如果我们的最终目标是获得字符串连接而不是数组，我们就可以用它代替 `listCombination(..)` ：

```
words.reduce( transducer( strConcat ), "" );
// 写点什么
```

Boom！ 这就是 transducing。

最后

深吸一口气，确实有很多要消化。

放空我们的大脑，让我们把注意力转移到如何在我们的程序中使用转换，而不是关心它的工作原理。

回想起我们之前定义的辅助函数，为清楚起见，我们重新命名一下：

```
var transduceMap = curry( function mapReducer(mapperFn,combinationFn){
    return function reducer(list,v){
        return combinationFn( list, mapperFn( v ) );
    };
} );

var transduceFilter = curry( function filterReducer(predicateFn,combinationFn){
    return function reducer(list,v){
        if (predicateFn( v )) return combinationFn( list, v );
        return list;
    };
} );
```

还记得我们这样使用它们：

```
var transducer = compose(
    transduceMap( strUppercase ),
    transduceFilter( isLongEnough ),
    transduceFilter( isShortEnough )
);
```

`transducer(..)` 仍然需要一个组合函数（如 `listCombination(..)` 或 `strConcat(..)`）来产生一个传递给 `reduce(..)` （连同初始值）的 transduce-reducer 函数。

但是为了更好的表达所有这些转换步骤，我们来做一个 `transduce(..)` 工具来为我们做这些步骤：

```
function transduce(transducer,combinationFn,initialValue,list) {
    var reducer = transducer( combinationFn );
    return list.reduce( reducer, initialValue );
}
```

这是我们的运行示例，梳理如下：

```
var transducer = compose(
    transduceMap( strUppercase ),
    transduceFilter( isLongEnough ),
    transduceFilter( isShortEnough )
);

transduce( transducer, listCombination, [], words );
// ["WRITTEN","SOMETHING"]

transduce( transducer, strConcat, "", words );
// 写点什么
```

不错，嗯！ 看到 `listCombination(..)` 和 `strConcat(..)` 函数可以互换使用组合函数了吗？

Transducers.js

最后，我们来说明我们运行的例子，使用sensors-js库（https://github.com/cognitect-labs/transducers-js ）：

```
var transformer = transducers.comp(
    transducers.map( strUppercase ),
    transducers.filter( isLongEnough ),
    transducers.filter( isShortEnough )
);

transducers.transduce( transformer, listCombination, [], words );
// ["WRITTEN","SOMETHING"]

transducers.transduce( transformer, strConcat, "", words );
// WRITTENSOMETHING
```

看起来几乎与上述相同。

注意： 上面的代码段使用 `transformers.comp(..)` ，因为这个库提供这个 API，但在这种情况下，我们从第 4 章的 `compose(..)` 也将产生相同的结果。换句话说，组合本身不是 transducing 敏感的操作。

该片段中的组合函数被称为 `transformer` ，而不是 `transducer` 。那是因为如果我们直接调用 `transformer(listCombination)` （或 `transformer(strConcat)` ），那么我们不会像以前那样得到一个直观的 transduce-reducer 函数。

`transducers.map(..)` 和 `transducers.filter(..)` 是特殊的辅助函数，可以将常规的断言函数或映射函数转换成适用于产生特殊变换对象的函数（里面包含了 reducer 函数）；这个库使用这些变换对象进行转换。此转换对象抽象的额外功能超出了我们将要探索的内容，请参阅该库的文档以获取更多信息。

由于 `transformer(..)` 产生一个变换对象，而不是一个典型的二元 transduce-reducer 函数，该库还提供 `toFn(..)` 来使变换对象适应本地数组的 `reduce(..)` 方法：

```
words.reduce(
    transducers.toFn( transformer, strConcat ),
    ""
);
// WRITTENSOMETHING
```

`into(..)` 是另一个提供的辅助函数，它根据指定的空/初始值的类型自动选择默认的组合函数：

```
transducers.into( [], transformer, words );
// ["WRITTEN","SOMETHING"]

transducers.into( "", transformer, words );
// WRITTENSOMETHING
```

当指定一个空数组 `[]` 时，内部的 `transduce(..)` 使用一个默认的函数实现，这个函数就像我们的 `listCombination(..)`。但是当指定一个空字符串 `“”` 时，会使用像我们的 `strConcat(..)` 这样的方法。这很酷！

如你所见，`transducers-js` 库使转换非常简单。我们可以非常有效地利用这种技术的力量，而不至于陷入定义所有这些中间转换器生产工具的繁琐过程中去。

总结

Transduce 就是通过减少来转换。更具体点，transduer 是可组合的 reducer。

我们使用转换来组合相邻的 `map(..)`、`filter(..)` 和 `reduce(..)` 操作。我们首先将 `map(..)` 和 `filter(..)` 表示为 `reduce(..)`，然后抽象出常用的组合操作来创建一个容易组合的一致的 reducer 生成函数。

transducing 主要提高性能，如果在延迟序列（异步 observables）中使用，则这一点尤为明显。

但是更广泛地说，transducing 是我们针对那些不能被直接组合的函数，使用的一种更具声明式风格的方法。否则这些函数将不能直接组合。如果使用这个技术能像使用本书中的所有其他技术一样用的恰到好处，代码就会显得更清晰，更易读！ 使用 transducer 进行单次 `reduce(..)` 调用比追踪多个 `reduce(..)` 调用更容易理解。

JavaScript 轻量级函数式编程

附录 B: 谦虚的 Monad

首先，我坦白：在开始写以下内容之前我并不太了解 Monad 是什么。我为了确认一些事情而犯了很多错误。如果你不相信我，去看看 这本书 Git 仓库 中关于本章的提交历史吧！

我在本书中囊括了所有涉及 Monad 的话题。就像我写书的过程一样，每个开发者在学习函数式编程的旅程中都会经历这个部分。

尽管其他函数式编程的著作差不多都把 Monad 作为开始，而我们却只对它做了简要说明，并基本以此结束本书。在轻量级函数式编程中我确实没有遇到太多需要仔细考虑 Monad 的问题，这就是本文更有价值的原因。但是并不是说 Monad 是没用的或者是不普遍的 —— 恰恰相反，它很有用，也很流行。

函数式编程界有一个小笑话，几乎每个人都不得不在他们的文章或者博客里写 Monad 是什么，把它拎出来写就像是一个仪式。在过去的几年里，人们把 Monad 描述为卷饼、洋葱和各种各样古怪的抽象概念。我肯定不会重蹈覆辙！

> 一个 Monad 仅仅是自函子 (endofunctor) 范畴中的一个 monoid

我们引用这句话来开场，所以把话题转到这个引言上面似乎是很合适的。可是才不会这样，我们不会讨论 Monad 、endofunctor 或者范畴论。这句引言不仅故弄玄虚而且华而不实。

我只希望通过我们的讨论，你不再害怕 Monad 这个术语或者这个概念了 —— 我曾经怕了很长一段时间 —— 并在看到该术语时知道它是什么。你可能，也只是可能，会正确地使用到它们。

类型

在函数式编程中有一个巨大的兴趣领域：类型论，本书基本上完全远离了该领域。我不会深入到类型论，坦白的说，我没有深入的能力，即使干了也吃力不讨好。

但是我要说，Monad 基本上是一个值类型。

数字 `42` 有一个值类型（number），它带有我们依赖的特征和功能。字符串 `"42"` 可能看起来很像，但是在编程里它有不同的用途。

在面向对象编程中，当你有一组数据（甚至是一个单独的离散值），并且想要给它绑上一些行为，那么你将创建一个对象或者类来表示 "type"。接着实例就成了该类型的一员。这种做法通常被称为 “数据结构”。

我将会非常宽泛的使用数据结构这个概念，而且我断定，当我们在编程中为一个特定的值定义一组行为以及约束条件，并且将这些特征与值一起绑定在一个单一抽象概念上时，我们可能会觉得很有用。这样，当我们在编程中使用一个或多个这种值的时候，它们的行为会自然的出现，并且会使它们更方便的工作。方便的是，对你的代码的读者来说，是更有描述性和声明性的。

Monad 是一种数据结构。是一种类型。它是一组使处理某个值变得可预测的特定行为。

回顾第 8 章，我们谈到了函子（functor）：包括一个值和一个用来对构成函子的数据执行操作的类 map 实用函数。Monad 是一个包含一些额外行为的函子（functor）。

松散接口

实际上，Monad 并不是单一的数据类型，它更像是相关联的数据类型集合。它是一种根据不同值的需要而用不同方式实现的接口。每种实现都是一种不同类型的 Monad。

例如，你可能阅读 "Identity Monad"、"IO Monad"、"Maybe Monad"、"Either Monad" 或其他形形色色的字眼。他们中的每一个都有基本的 Monad 行为定义，但是它根据每个不同类型的 Monad 用例来继承或者重写交互行为。

可是它不仅仅是一个接口，因为它不只是使对象成为 Monad 的某些 API 方法的实现。对这些方法的交互的保障是必须的，是 monadic 的。这些众所周知的常量对于使用 Monad 提高可读性是至关重要的；另外，它是一个特殊的数据结构，读者必须全部阅读才能明白。

事实上，这些 Monad 方法的名字和真实接口授权的方式甚至没有一个统一的标准；Monad 更像是一个松散接口。有些人称这些方法为 `bind(..)`,有些称它为 `chain(..)`，还有些称它为 `flatMap(..)`，等等。

所以，Monad 是一个对象数据结构，并且有充足的方法（几乎任何名称或排序），至少满足了 Monad 定义的主要行为需求。每一种 Monad 都基于最少数量的方法来进行不同的扩展。但是，因为它们在行为上都有重叠，所以一起使用两种不同的 Monad 仍然是直截了当和可控的。

从某种意义上说，Monad 更像是接口。

Maybe

在函数式编程中，像 Maybe 这样涵盖 Monad 是很普遍的。事实上，Maybe Monad 是另外两个更简单的 Monad 的搭配：Just 和 Nothing。

既然 Monad 是一个类型，你可能认为我们应该定义 `Maybe` 作为一个要被实例化的类。这虽然是一种有效的方法，但是它引入了 `this` 绑定的问题，所以在这里我不想讨论；相反，我打算使用一个简单的函数和对象的实现方式。

以下是 Maybe 的最简单的实现：

```
var Maybe = { Just, Nothing, of/* 又称：unit，pure */: Just };

function Just(val) {
    return { map, chain, ap, inspect };

    // *********************

    function map(fn) { return Just( fn( val ) ); }
    // 又称：bind, flatMap
    function chain(fn) { return fn( val ); }
    function ap(anotherMonad) { return anotherMonad.map( val ); }

    function inspect() {
        return `Just(${ val })`;
    }
}

function Nothing() {
    return { map: Nothing, chain: Nothing, ap: Nothing, inspect };

    // *********************

    function inspect() {
        return "Nothing";
    }
}
```

注意： `inspect(..)` 方法只用于我们的示例中。从 Monad 的角度来说，它并没有任何意义。

如果现在大部分都没有意义的话，不要担心。我们将会更专注的说明我们可以用它做什么，而不是过多的深入 Monad 背后的设计细节和理论。

所有的 Monad 一样，任何含有 `Just(..)` 和 `Nothing()` 的 Monad 实例都有 `map(..)` 、 `chain(..)` （也叫 `bind(..)` 或者 `flatMap(..)` ）和 `ap(..)` 方法。这些方法及其行为的目的在于提供多个 Monad 实例一起工作的标准化方法。你将会注意到，无论 `Just(..)` 实例拿到的是怎样的一个 `val` 值， `Just(..)` 实例都不会去改变它。所有的方法都会创建一个新的 Monad 实例而不是改变它。

Maybe 是这两个 Monad 的结合。如果一个值是非空的，它是 `Just(..)` 的实例；如果该值是空的，它则是 `Nothing()` 的实例。注意，这里由你的代码来决定 "空" 的意思，我们不做强制限制。下一节会详细介绍这一点。

但是 Monad 的价值在于不论我们有 `Just(..)` 实例还是 `Nothing()` 实例，我们使用的方法都是一样的。 `Nothing()` 实例对所有的方法都有空操作定义。所以如果 Monad 实例出现在 Monad 操作中，它就会对 Monad 操作起短路（short-circuiting）作用。

Maybe 这个抽象概念的作用是隐式地封装了操作和无操作的二元性。

与众不同的 Maybe

JavaScript Maybe Monad 的许多实现都包含 `null` 和 `undefined` 的检查（通常在 `map(..)` 中），如果是空的话，就跳过该 Monad 的特性行为。事实上，Maybe 被声称是有价值的，因为它自动地封装了空值检查得以在某种程度上短路了它的特性行为。

这是 Maybe 的典型说明：

```
// 代替不稳定的 `console.log( someObj.something.else.entirely )`：

Maybe.of( someObj )
.map( prop( "something" ) )
.map( prop( "else" ) )
.map( prop( "entirely" ) )
.map( console.log );
```

换句话说，如果我们在链式操作中的任何一环得到一个 `null` 或者 `undefined` 值，Maybe 会智能的切换到空操作模式 —— 它现在是一个 `Nothing()` Monad 实例！ —— 把剩余的链式操作都停止掉。如果一些属性丢失或者是空的话，嵌套的属性访问能安全的抛出 JS 异常。这是非常酷的而且很实用。

但是，我们这样实现的 Maybe 不是一个纯 Monad。

Monad 的核心思想是，它必须对所有的值都是有效的，不能对值做任何检查 —— 甚至是空值检查。所以为了方便，这些其他的实现都是走的捷径。这是无关紧要的。但是当学习一些东西的时候，你应该先学习它的最纯粹的形式，然后再学习更复杂的规则。

我早期提供的 Maybe Monad 的实现不同于其他的 Maybe，就是它没有空置检查。另外，我们将 `Maybe` 作为 `Just(..)` 和 `Nothing()` 的非严格意义上的结合。

等一下，如果我们没有自动短路，那 Maybe 是怎么起作用的呢？！？这似乎就是它的全部意义。

不要担心，我们可以从外部提供简单的空值检查，Maybe Monad 其他的短路行为也还是可以很好的工作的。你可以在之前做一些 `someObj.something.else.entirely` 属性嵌套，但是我们可以做的更“正确”：

```
function isEmpty(val) {
    return val === null || val === undefined;
}

var safeProp = curry( function safeProp(prop,obj){
    if (isEmpty( obj[prop] )) return Maybe.Nothing();
    return Maybe.of( obj[prop] );
} );

Maybe.of( someObj )
.chain( safeProp( "something" ) )
.chain( safeProp( "else" ) )
.chain( safeProp( "entirely" ) )
.map( console.log );
```

我们设计了一个用于空值检查的 `safeProp(..)` 函数，并选择了 `Nothing()` Monad 实例。或者把值包装在 `Just(..)` 实例中（通过 `Maybe.of(..)`）。然后我们用 `chain(..)` 替代 `map(..)`，它知道如何“展开” `safeProp(..)` 返回的 Monad。

当遇到空值的时候，我们得到了一连串相同的短路。只是我们把这个逻辑从 Maybe 中排除了。

不管返回哪种类型的 Monad，我们的 `map(..)` 和 `chain(..)` 方法都有不变且可预测的反馈，这就是 Monad，尤其是 Maybe Monad 的好处。这难道不酷吗？

Humble

现在我们对 Maybe 和它的作用有了更多的了解，我将会在它上面加一些小的改动 —— 我将通过设计 Maybe + Humble Monad 来添加一些转折并且加一些诙谐的元素。从技术上来说，`Humble(..)` 并不是一个 Monad，而是一个产生 Maybe Monad 实例的工厂函数。

Humble 是一个使用 Maybe 来跟踪 `egoLevel` 数字状态的数据结构包装器。具体来说，`Humble(..)` 只有在他们自身的水平值足够低（少于 `42`）到被认为是 Humble 的时候才会执行生成的 Monad 实例；否则，它就是一个 `Nothing()` 空操作。这听起来真的和 Maybe 很像！

这是一个 Maybe + Humble Monad 工厂函数：

```
function Humble(egoLevel) {
    // 接收任何大于等于 42 的数字
    return !(Number( egoLevel ) >= 42) ?
        Maybe.of( egoLevel ) :
        Maybe.Nothing();
}
```

你可能会注意到，这个工厂函数有点像 `safeProp(..)`，因为，它使用一个条件来决定是选择 Maybe 的 `Just(..)` 还是 `Nothing()`。

让我们来看一个基础用法的例子：

```
var bob = Humble( 45 );
var alice = Humble( 39 );

bob.inspect();                          // Nothing
alice.inspect();                        // Just(39)
```

如果 Alice 赢得了一个大奖，现在是不是在为自己感到自豪呢？

```
function winAward(ego) {
    return Humble( ego + 3 );
}

alice = alice.chain( winAward );
alice.inspect();                        // Nothing
```

`Humble( 39 + 3 )` 创建了一个 `chain(..)` 返回的 `Nothing()` Monad 实例，所以现在 Alice 不再有 Humble 的资格了。

现在，我们来用一些 Monad ：

```
var bob = Humble( 41 );
var alice = Humble( 39 );

var teamMembers = curry( function teamMembers(ego1,ego2){
    console.log( `Our humble team's egos: ${ego1} ${ego2}` );
} );

bob.map( teamMembers ).ap( alice );
// Humble 队列：41 39
```

由于 `teamMembers(..)` 是柯里化的，`bob.map(..)` 的调用传入了 `bob` 自身的级别（`41`），并且创建了一个被其余的方法包装的 Monad 实例。在 这个 Monad 中调用的 `ap(alice)` 调用了 `alice.map(..)`，并且传递给来自 Monad 的函数。这样做的效果是，Monad 的值已经提供给了 `teamMembers(..)` 函数，并且把显示的结果给打印了出来。

然而，如果一个 Monad 或者两个 Monad 实际上是 `Nothing()` 实例（因为它们本身的水平值太高了）：

```
var frank = Humble( 45 );

bob.map( teamMembers ).ap( frank );

frank.map( teamMembers ).ap( bob );
```

`teamMembers(..)` 永远不会被调用（也没有信息被打印出来），因为，`frank` 是一个 `Nothing()` 实例。这就是 Maybe monad 的作用，我们的 `Humble(..)` 工厂函数允许我们根据自身的水平来选择。赞！

Humility

再来一个例子来说明 Maybe + Humble 数据结构的行为：

```
function introduction() {
    console.log( "I'm just a learner like you! :)" );
}

var egoChange = curry( function egoChange(amount,concept,egoLevel) {
    console.log( `${amount > 0 ? "Learned" : "Shared"} ${concept}.` );
    return Humble( egoLevel + amount );
} );

var learn = egoChange( 3 );

var learner = Humble( 35 );

learner
.chain( learn( "closures" ) )
.chain( learn( "side effects" ) )
.chain( learn( "recursion" ) )
.chain( learn( "map/reduce" ) )
.map( introduction );
// 学习闭包
// 学习副作用
// 歇息递归
```

不幸的是，学习过程看起来已经缩短了。我发现学习一大堆东西而不和别人分享，会使自我太膨胀，这对你的技术是不利的。

让我们尝试一个更好的方法：

```
var share = egoChange( -2 );

learner
.chain( learn( "closures" ) )
.chain( share( "closures" ) )
.chain( learn( "side effects" ) )
.chain( share( "side effects" ) )
.chain( learn( "recursion" ) )
.chain( share( "recursion" ) )
.chain( learn( "map/reduce" ) )
.chain( share( "map/reduce" ) )
.map( introduction );
// 学习闭包
// 分享闭包
// 学习副作用
// 分享副作用
// 学习递归
// 分享递归
// 学习 map/reduce
// 分享 map/reduce
// 我只是一个像你一样的学习者 :)
```

在学习中分享。是学习更多并且能够学的更好的最佳方法。

总结

说了这么多，那什么是 Monad ？

Monad 是一个值类型，一个接口，一个有封装行为的对象数据结构。

但是这些定义中没有一个是有用的。这里尝试做一个更好的解释：Monad 是一个用更具有声明式的方式围绕一个值来组织行为的方法。

和这本书中的其他部分一样，在有用的地方使用 Monad，不要因为每个人都在函数式编程中讨论他们而使用他们。Monad 不是万金油，但它确实提供了一些有用的实用函数。

JavaScript 轻量级函数式编程

附录 C：函数式编程函数库

如果您已经从头到尾通读了此书，请花一分钟的时间停下来回顾一下从第 1 章到现在的收获。相当漫长的一段旅程，不是吗？希望您已经收获了大量新知识，并用函数式的方式思考你的程序。

在本书即将完结时，我想给你提供一些关于使用官方函数式编程函数库的快速指南。注意这并不是一个详细的文档，而是将你在结束“轻量级函数式编程”后进军真正的函数式编程时应该注意的东西快速梳理一下。

如果有可能，我建议你不要做重新造轮子这样的事情。如果你找到了一个能满足你需求的函数式编程函数库，那么用它就对了。只有在你实在找不到合适的库来应对你面临的问题时，才应该使用本书提供的辅助实用函数 —— 或者自己造轮子。

目录

在本书第 1 章曾列出了一个函数式编程库的列表，现在我们来扩展这个列表。我们不会涉及所有的库（它们之中有许多重复的内容），但下面这些你应该有所关注：

- Ramda：通用函数式编程实用函数
- Sanctuary：函数式编程类型 Ramda 伴侣
- lodash/fp：通用函数式编程实用函数
- functional.js：通用函数式编程实用函数
- Immutable：不可变数据结构
- Mori：（受到 ClojureScript 启发）不可变数据结构
- Seamless-Immutable：不可变数据助手
- tranducers-js：数据转换器
- monet.js：Monad 类型

上面的列表只列出了所有函数式编程库的一小部分，并不是说没有在列表中列出的库就不好，也不是说列表中列出的就是最佳选择，总之这只是 JavaScript 函数式编程世界中的一瞥。您可以前往这里查看更完整的函数式编程资源。

Fantasy Land（又名 FL）是函数式编程世界中十分重要的学习资源之一，与其说它是一个库，不如说它是一本百科全书。

Fantasy Land 不是一份为初学者准备的轻量级读物，而是一个完整而详细的 JavaScript 函数式编程路线图。为了尽可能提升互通性，FL 已经成为 JavaScript 函数式编程库遵循的实际标准。

Fantasy Land 与“轻量级函数式编程”的概念相反，它以火力全开的姿态进军 JavaScript 的函数式编程世界。也就是说，当你的能力超越本书时，FL 将会成为你接下来前进的方向。我建议您将其保存在收藏夹中，并在您使用本书的概念进行至少 6 个月的实战练习之后再回来。

Ramda (0.23.0)

摘自 Ramda 文档:

> Ramda 函数自动地被柯里化。
>
> Ramda 函数的参数经过优化，更便于柯里化。需要被操作的数据往往放在最后提供。

我认为合理的设计是 Ramda 的优势之一。值得注意的是，Ramda 的柯里化形式（似乎大多数的库都是这种形式）是我们在第 3 章中讨论过的“松散柯里化”。

第 3 章的最后一个例子 —— 我们定义无值（point-free）工具函数 `printIf()` —— 可以在 Ramda 中这样实现：

```
function output(msg) {
    console.log( msg );
}

function isShortEnough(str) {
    return str.length <= 5;
}

var isLongEnough = R.complement( isShortEnough );

var printIf = R.partial( R.flip( R.when ), [output] );

var msg1 = "Hello";
var msg2 = msg1 + " World";

printIf( isShortEnough, msg1 );            // Hello
printIf( isShortEnough, msg2 );

printIf( isLongEnough, msg1 );
printIf( isLongEnough, msg2 );            // Hello World
```

与我们在第 3 章中的实现相比有几处不同：

- 我们使用 `R.complement(..)` 而不是 `not(..)` 在 `isShortEnough(..)` 周围新建一个否定函数 `isLongEnough(..)` 。

- 使用 `R.flip(..)` 而不是 `reverseArgs(..)` 函数，值得一提的是，`R.flip(..)` 仅交换头两个参数，而 `reverseArgs(..)` 会将所有参数反向。在这种情景下，`flip(..)` 更加方便，所以我们不再需要使用 `partialRight(..)` 或其他投机取巧的方式进行处理。
- `R.partial(..)` 所有的后续参数以单个数组的形式存在。
- 因为 Ramda 使用松散柯里化，因此我们不需要使用 `R.uncurryN(..)` 来获得一个包含所有参数的 `printIf(..)`。如果我们这样做了，就相当于使用 `R.uncurryN(2, ..)` 包裹 `R.partial(..)` 进行调用，这是完全没有必要的。

Ramda 是一个受欢迎的、功能强大的库。如果你想要在你的代码中实践 FP，从 Ramda 开始是个不错的选择。

Lodash/fp (4.17.4)

Lodash 是整个 JS 生态系统中最受欢迎的库。Lodash 团队发布了一个“FP 友好”的 API 版本 —— "lodash/fp"。

在第 8 章中，我们讨论了合并独立列表操作（`map(..)`、`filter(..)` 以及 `reduce(..)`）。使用“lodash/fp”时，你可以这样做：

```
var sum = (x,y) => x + y;
var double = x => x * 2;
var isOdd = x => x % 2 == 1;

fp.compose( [
    fp.reduce( sum )( 0 ),
    fp.map( double ),
    fp.filter( isOdd )
] )
( [1,2,3,4,5] );                    // 18
```

与我们所熟知的 `_.` 命名空间前缀不同，“lodash/fp”将 `fp.` 定义为其命名空间前缀。我发现一个很有用的区别，就是 `fp.` 比 `_.` 更容易识别。

注意 `fp.compose(..)`（在常规 lodash 版本中又名 `_.flowRight(..)`）接受一个函数数组，而不是独立的函数作为参数。

lodash 拥有良好的稳定性、广泛的社区支持以及优秀的性能，是你探索 FP 世界时的坚实后盾。

Mori (0.3.2)

在第 6 章中，我们已经快速浏览了一下 Immutable.js 库，该库可能是最广为人知的不可变数据结构库了。

让我们来看一下另一个流行的库：Mori。Mori 设计了一套与众不同（从表面上看更像函数式编程）的 API：它使用独立的函数而不直接在值上操作。

```
var state = mori.vector( 1, 2, 3, 4 );

var newState = mori.assoc(
    mori.into( state, Array.from( {length: 39} ) ),
    42,
    "meaning of life"
);

state === newState;                          // false

mori.get( state, 2 );                      // 3
mori.get( state, 42 );                      // undefined

mori.get( newState, 2 );                  // 3
mori.get( newState, 42 );                  // "meaning of life"

mori.toJs( newState ).slice( 1, 3 );    // [2,3]
```

这是一个指出关于 Mori 的一些有趣的事情的例子：

- 使用 `vector` 而不是 `list`（你可能会想用的），主要是因为文档说它的行为更像 JavaScript 中的数组。
- 不能像在操作原生 JavaScript 数组那样在任意位置设置值，在 vector 结构中，这将会抛出异常。因此我们必须使用 `mori.into(..)`，传入一个合适长度的数组来扩展 vector 的长度。在上例中，vector 有 43 个可用位置（4 + 39），所以我们可以在最后一个位置（索引为 42）上写入 `"meaning of life"` 这个值。
- 使用 `mori.into(..)` 创建一个较大的 vector，再用 `mor.assoc(..)` 根据这个 vector 创建另一个 vector 的做法听起来效率低下。但是，不可变数据结构的好处在于数据不会进行克隆，每次“改变”发生，新的数据结构只会追踪其与旧数据结构的不同之处。

Mori 受到 ClojureScript 极大的启发。如果您有 ClojureScript 编程经验，那您应该对 Mori 的 API 感到非常熟悉。由于我没有这种编程经验，因此我感觉 Mori 中的方法名有点奇怪。

但相比于在数据上直接调用方法，我真的很喜欢调用独立方法这样的设计。Mori 还有一些自动返回原生 JavaScript 数组的方法，用起来非常方便。

总结

JavaScript 不是作为函数式编程语言来特别设计的。不过其自身的确拥有很多对函数式编程非常友好基础语法（例如可作为变量的函数、闭包等）。本章提及的库将使你更方便的进行函数式编程。

有了本书中函数式编程概念的武装，相信你已经准备好开始处理现实世界的代码了。找一个优秀的函数式编程库来用，然后练习，练习，再练习。

就是这样了。我已经将我目前所知道的知识分享给你了。我在此正式认证您为“JavaScript 轻量级函数式编程”程序员！好了，是时候结束我们一起学习 FP 这部分的“章节”了，但我的学习之旅还将继续。我希望，你也是！